Practical TMP:

A C++17 Compile Time Register Machine

TMP is Turing Complete

Template meta programming is Turing Complete.

A brief history:

- 1994 demonstration of a prime number generator by Erwin Unruh.
- Articles: "C++ Templates are Turing Complete" (Veldhuizen)
- Books: C++ Templates: The Complete Guide (Vandevoorde; Josuttis)
- Meta programming Libraries: Boost, among others.

Is TMP practical?

Template meta programming is practical:

- Classic example: Loop unrolling.
- Paradigms: CRTP, SFINAE, etc.
- Compile Time Regular Expressions by Hana Dusíková.

What about a TMP Turing machine?

A TMP Register Machine

With a bit of care, a TMP Turing machine can be practical too!

The solution to this problem is framed as seven *bottlenecks*.

Theoretical: (what will prevent TMP Turing from being possible?)

1. A Stack Machine
2. Continuation Passing

A TMP Register Machine

- Practical: (what will prevent TMP Turing from being reasonable?)
  3. The Nesting Depth Problem
  4. Interoperability
  5. Organizational Design
  6. Debugging
  7. Performance

These bottlenecks inform the remainder of this talk.

Theory

1. A Stack Machine

A Stack Machine

- There are several different ways to implement a function or machine to be Turing complete. Which approach makes the most sense in terms of meta programming?

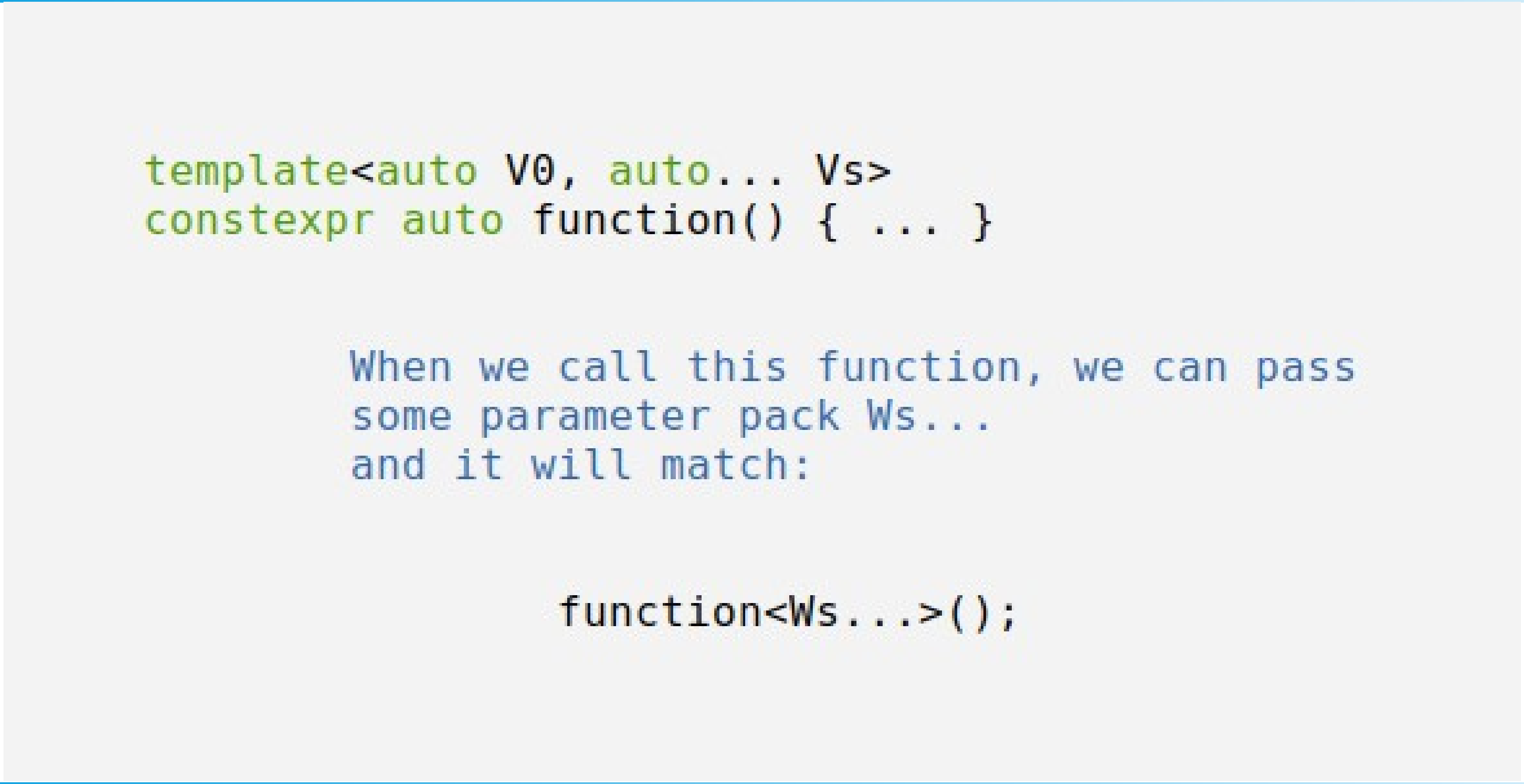
- I use variadic packs (Vs...) to achieve this goal, which is possible because Automata Theory tells us that a finite state machine equipped with a two stack memory system is Turing complete.

A Stack Machine

- What’s the relationship between stacks and variadic packs?

- A side effect of template parameter resolution is the ability to pattern match from the front of a variadic pack.
- This means we can interpret a pack as a stack, and push and pop from the front:

A Stack Machine



```
template<auto V0, auto... Vs>
constexpr auto function() { ... }

        When we call this function, we can pass
        some parameter pack Ws...
        and it will match:

                function<Ws...>();
```

A Stack Machine

- Thus, if we want to implement a Turing machine using parameter packs, we need at least two of them. This is a problem as template scopes only allow directly for one.
- This is where constexpr functions come in:

```
template<auto... Stack1, auto... Stack2>
constexpr auto function(auto_pack<Stack2...>) { ... }
```

A Stack Machine

- The reason two stacks are sufficient to build a Turing machine is because they in effect allow random access reading and writing of their memory states (similar to a “tape machine”):

```
template<
        auto... Stack1_Rest,
        auto... Stack1_Front, auto... Stack2
>
constexpr auto machine(auto_pack<Stack1_Front..., Stack2...>) { ... }
```

A Stack Machine

- Is this all?
- There’s more: We’ve described the memory component of these machines, but we still require our constexpr function to be a finite state machine.
- In Automata Theory this is called a “transition function”.

A Stack Machine

- In terms of Register Machine Theory this is called a controller. Our stack machine's controller will be added to the front of the template stack:

```
template<
        MACHINE_CONTROLLER, auto... Stack1,
        auto... Stack2
>
constexpr auto machine(auto_pack<Stack2...>) { ... }
```

A Stack Machine

- What are the controller requirements?
- A register machine controller is made up labels, which are themselves made up of instructions---which forms the basis of real world assembly languages.
- Thus, our stack machine also requires a *controller language*.
- Although this language can be modelled off of any existing assembly language, I have decided to use a modified form of the language provided in Chapter 5 of the classic text “Structure and Interpretation of Computer Programs” (SICP):

A Stack Machine

```
(assign <register-name> (reg <register-name>))

(assign <register-name> (const <constant-value>))

(assign <register-name> (op <operation-name>) <input_1> ... <input_n>)

(perform (op <operation-name>) <input_1> ... <input_n>)

(test (op <operation-name>) <input_1> ... <input_n>)

(branch (label <label-name>))

(goto (label <label-name>))

(assign <register-name> (label <label-name>))          (save <register-name>)

(goto (reg <register-name>))                           (restore <register-name>)
```

2. Continuation Passing

Continuation Passing

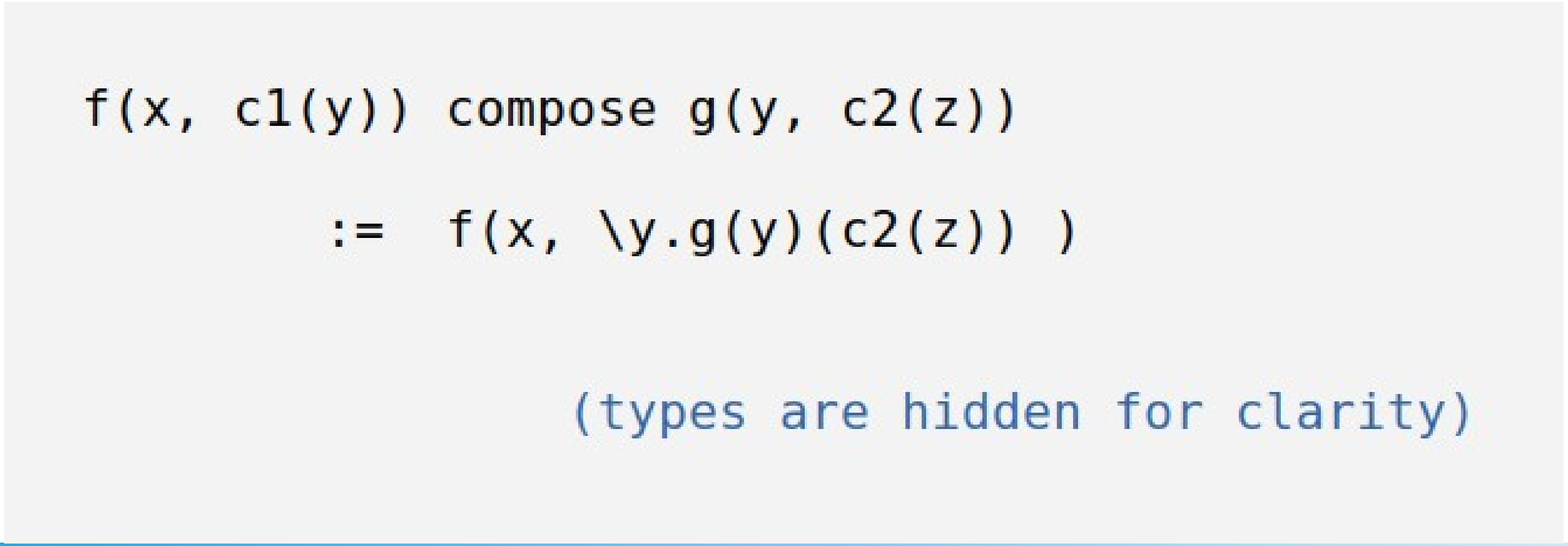
- Now that we have our stack machine, how do we actually go from one state of the machine to the next?

- Continuation passing comes to the rescue!

- In particular, Category Theory tells us that continuation passing is a monad which is what we need. Don’t worry, We won’t be going into that theory here, but it’s there if you need it.

Continuation Passing

The important idea for us is that of continuation passing *composition*:



```
f(x, c1(y)) compose g(y, c2(z))

        :=  f(x, \y.g(y)(c2(z)) )


                        (types are hidden for clarity)
```

Continuation Passing

- What does this mean for us?

- We interpret each controller instruction as being symbolic of its own continuation passing function.
- This means that our stack machines will perform their intended instructional behaviour, but instead of returning directly they pass their results to the next stack machine:

Continuation Passing

```
template<
        MACHINE_CONTROLLER, auto... Stack1,
        auto... Stack2
>
constexpr auto machine(auto_pack<Stack2...> Heap) {

        return next_machine
        <
                MACHINE_CONTROLLER, Stack1...

        >(Heap);
}
```

Continuation Passing

- For practical reasons, rather than giving these machines their own unique names as constexpr functions, we enclose them in a templated struct:

```
template<>
struct machine<MN::(((name))), (((note)))> {

        template<
                MACHINE_CONTROLLER, auto... Stack1,
                auto... Stack2
        >
        static constexpr auto result(auto_pack<Stack2...>) { ... }
};
```

Continuation Passing

- Also note: In theory we only need one template parameter indexing the *name* of the machine, but in practice I’ve found it’s useful to have a second parameter called the *note*, which makes it easier to dispatch to variations or optimized versions.

Continuation Passing

- How do we know what the next machine is?

- All of that information is held in the controller. In which case, we have two additional requirements:
  1. An index telling us where we currently are within the controller.
  2. A dispatch function that knows how to get to the next instruction from the current index.

Continuation Passing

```
template<>
struct machine<MN::(((name))), (((note)))> {
        template<
                auto controller, auto index,
                auto... Stack1, auto... Stack2
        >
        static constexpr auto result(auto_pack<Stack2...> Heap) {

                return machine<
                        next_name(controller, index),
                        next_note(controller, index)
                >::template result<
                        controller, next_index(controller, index),
                        Stack1...

                >(Heap);
        }};
```

Continuation Passing

- So we need one dispatch, and one index to continuation pass.
- Is that all?
- In an ideal world, yes. In theory we could make our dispatch to be generic, and our index to be anything: An integer, or an array for example.

Continuation Passing

- In practice (for performance reasons we will get to later), it’s better to have several dispatches, and two indices:

```
template<
        typename dispatches, auto controller, auto index1, auto index2,
        auto... Stack1, auto... Stack2
>
static constexpr auto result(auto_pack<Stack2...>) { ... }
```

Continuation Passing

```
template<>
struct machine<MN::(((name))), (((note)))> {
        template<
                typename dispatches, auto controller, auto index1, auto index2,
                auto... Stack1, auto... Stack2
        >
        static constexpr auto result(auto_pack<Stack2...> Heap) {

                return machine<
                        dispatches::next_name(controller, index1, index2),
                        dispatches::next_note(controller, index1, index2)
                >::template result<
                        dispatches, controller,
                        dispatches::next_index1(controller, index1, index2),
                        dispatches::next_index2(controller, index1, index2),
                        Stack1...

                >(Heap);
        }};
```

Continuation Passing

- We still have a bit more to add to this general pattern, so let’s simplify our current variables before we continue:

Continuation Passing

```
template<>
struct machine<MN::(((name))), (((note)))> {
        template<
                typename n, auto c, auto i, auto j,
                auto... Stack1, auto... Stack2
        >
        static constexpr auto result(auto_pack<Stack2...> Heap) {

                return machine<
                        n::next_name(c, i, j),
                        n::next_note(c, i, j)
                >::template result<
                        n, c,
                        n::next_index1(c, i, j),
                        n::next_index2(c, i, j),
                        Stack1...

                >(Heap);
        }};
```

Practical

3. The Nesting Depth Problem

The Nesting Depth Problem

- The nesting depth problem is a result of using continuation passing.
- The issue in particular is the unbroken chain of calls that we make to machine after machine: Each call subtracts from the total allowable depth, which runs out quickly as compilers set fairly small default limits:
  - GCC – 900   (v8.4.0 on my machine)
  - Clang – 512 (v6.0.0 on my machine)

The Nesting Depth Problem

- So how do we mitigate this problem?

- Trampolines are the answer!
- Trampolining is where we return from our continuation passing with an intermediate result.
- We trampoline before the nesting depth runs out which allows us to reset the depth back to zero and try again until we’re done.

The Nesting Depth Problem

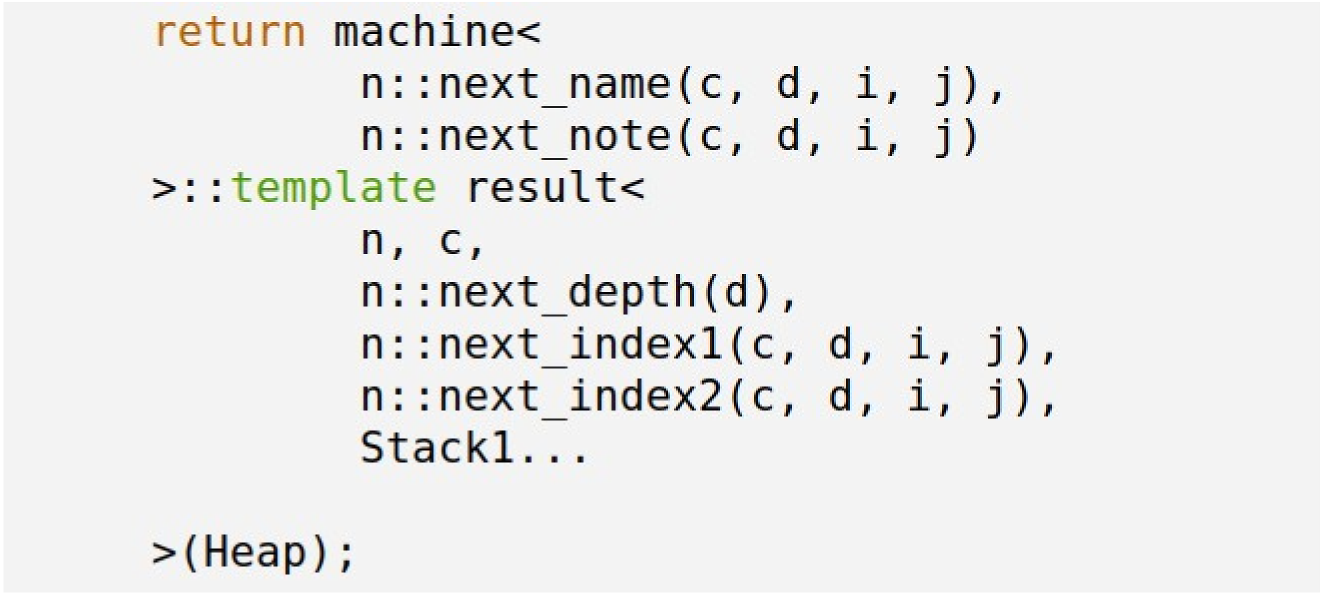
- How do we apply this our machines?

- We add in one more index variable to the controller:

```
template<
        typename n, auto c, auto depth, auto i, auto j,
        auto... Stack1, auto... Stack2
>
static constexpr auto result(auto_pack<Stack2...>) { ... }
```

The Nesting Depth Problem

- This also means we have to update our dispatch inventory to include one more function:



```
return machine<
        n::next_name(c, d, i, j),
        n::next_note(c, d, i, j)
>::template result<
        n, c,
        n::next_depth(d),
        n::next_index1(c, d, i, j),
        n::next_index2(c, d, i, j),
        Stack1...

>(Heap);
```

The Nesting Depth Problem

- The way these dispatches now work is if our depth counter is greater than zero, we continue to the next machine, otherwise we dispatch to a specialized machine (I’ve named it “pause”) which caches and returns the intermediate result.

- It should be noted that in practice it’s best to set our initial counter depth to be smaller than the compiler’s. For example if clang is 512, then we can set ours to 500.

The Nesting Depth Problem

So if our initial nesting depth is 500, what is the total trampolining depth?

I currently have trampolining implemented such that it subtracts 2 depths each time it trampolines. If you do the math,

500 + 498 + 496 + ... + 4 + 2 you get:

= 62,750 (total depths)

The Nesting Depth Problem

- Is that enough?

- For many of the simpler register machines it should be, but I have at least one test case where it’s not.
- When it’s not, it’s quite straightforward to then write a second trampolining function which trampolines the first.
- Such “higher order” trampolining can then be done as needed.

4. Interoperability

Interoperability

- What is meant by *interoperability?*
- Inter + Operations = Composability.
- This was touched on already with continuation passing, but it’s one thing to pass our data to the next function, it’s another if we need to call an internal function to perform our current instruction.

Interoperability

- The bottleneck comes from our nesting depth counter which is meant to tell us how many depths we have left, but if we call a non-machine function internally as a helper to our machine, we might lose track of how many depths actually remain.
- So we have a constraint: No internal function calls are allowed unless they have a known fixed depth (more on this later).

- The solution to this problem then is actually to extend our heap to become its own stack:

Interoperability

```
template<
        typename n, auto c, auto d, auto i, auto j,
        auto... Stack, typename... Heaps
>
static constexpr auto result(Heaps... Hs) {

        return machine<
                n::next_name(c, d, i, j),
                n::next_note(c, d, i, j)
        >::template result<
                n, c,
                n::next_depth(d),
                n::next_index1(c, d, i, j),
                n::next_index2(c, d, i, j),
                Stack...

        >(Hs...);
```

Interoperability

- Rather than calling independent helper functions, we now restrict ourselves to calling other machines which have their own controllers. We do this by caching our current controller as a heap (effectively pausing it).
- Doing things this way means we can not only pass the counter to the helper function that we call, we can still return to our original machine after.
- In summary: The only *internal functions* allowed (besides our fixed depth functions) are other machines.

Interoperability

- Is that it for interoperability?

- No it is not. There is actually a second bottleneck to consider: The *typename vs auto* problem:

```
template<
        MACHINE_CONTROLLER,
        typename... Stack, // instead of auto... Stack,
                typename... Heaps
>
static constexpr auto result(Heaps... Hs) { ... }
```

Interoperability

- Should we be using typename packs instead of auto?

- It’s a reasonable idea if you think about it: Constexpr functions already exist to do numeric calculations, so if we were to build register machine functions at all they would probably be used for *typename* manipulations.

- Fortunately we don’t have to make this decision!

Interoperability

- There is a *typename auto equivalence* which not only allows us to encode auto values as typenames (such as):

```
template<auto value>
struct value_cached_as_type { };

       using x = value_cached_as_type<int(5)>;
```

- But we can also encode typenames as auto values:

```
template<typename Type>
constexpr void type_cached_as_value(Type) { }

       constexpr auto y = type_cached_as_value<int>;
```

Interoperability

- The idea is to hide the typename as the input type of a void function, which can later be recovered through pattern matching.
- There are a few subtleties to take care of in this approach, but it works because function pointers can be passed as template parameters.
- In any case, because of this we really only need auto packs for our machines.

5. Organizational Design

Organizational Design

Although I didn’t mention it earlier, we actually now have the final form for our stack machines:

```
template<>
struct machine<MN::(((name))), (((note)))> {

        template<
                typename n, auto c, auto d, auto i, auto j,
                auto... Stack, typename... Heaps
        >
        static constexpr auto result(Heaps... Hs) {

                return machine<
                        n::next_name(c, d, i, j),
                        n::next_note(c, d, i, j)

                >::template result<
                        n, c,
                        n::next_depth(d),
                        n::next_index1(c, d, i, j),
                        n::next_index2(c, d, i, j),
                        Stack...

                >(Hs...);
        }};
```

Organizational Design

- Now that we have our general design for a single machine, there are some questions to bring up:
  1. Where do we go from here?
  2. How best to build a register machine library?
  3. Which specific machines do we choose to make?
  4. How do we organize our machines in this library?

Organizational Design

- This is organizational design, but why is it a bottleneck?
- Without getting into the specifics: How a library is organized ends up effecting maintenance, performance, and the ability to debug.
- In turn, I have devised the following hierarchy of dependencies for my own library:

Organizational Design

- Hierarchy:

1. Block machines	(atomics 1: pop, push, fold, etc.)
2. Variadic machines	(atomics 2: pop, push, fold, etc.)
3. Permutatic machines	(linear 1: stack/heap operators)
4. Distributic machines	(linear 2: erase, insert, replace)
5. Near linear machines	(1-cycle loops: lift, stem, cycle)
6. Register machines	(branch, goto, save, restore)

Organizational Design

- Some stats about this hierarchy:

- Within my current library there are 60 different machine names.
- There are 195 distinct machines altogether, but two of the modules { block, permutatic } actually account for 158 of these machines---most of which are optimized variants.

  (the optimized forms are coded with macros)

6. Debugging

Debugging

- Debugging is a practical bottleneck for any TMP project.

- I think this claim is self-explanatory and generally accepted by anyone who has done TMP, but just to give an idea of the problem, here is an example of a typical template error message:

  (gcc)

Debugging

```
source/3_variadic_machines.hpp:174:37:   required from ‘static constexpr auto machine_space::machi
ne<18>::result(Heap0, Heap1, Heaps ...) [with n = machine_space::PD; auto c = machine_space::f_arr
ay<unsigned char (*)(unsigned char), machine_space::f_array<unsigned char, 3>, machine_space::f_ar
ray<unsigned char, 3, 31, 1, 0>, machine_space::f_array<unsigned char, 2, 18, 0>, machine_space::f
_array<unsigned char, 3, 14, 0, 0> >; auto d = 498; auto i = 2; auto j = 0; auto pos = 4; auto ...
Vs = {square<int>, 5, square<int>, meta_programming::multiply_by<int, 2>, meta_programming::add_by
<int, 1>}; Heap0 = void (*)(meta_programming::auto_pack<meta_programming::type_map<S_do_compose*>
>*); Heap1 = void (*)(meta_programming::auto_pack<>*); Heaps = {}]’
source/4_permutatic_machines.hpp:107:3:   required from ‘static constexpr auto machine_space::mach
ine<31, 1>::result(void (*)(meta_programming::auto_pack<Ws ...>*), Heaps ...) [with n = machine_sp
ace::PD; auto c = machine_space::f_array<unsigned char (*)(unsigned char), machine_space::f_array<
unsigned char, 3>, machine_space::f_array<unsigned char, 3, 31, 1, 0>, machine_space::f_array<unsi
gned char, 2, 18, 0>, machine_space::f_array<unsigned char, 3, 14, 0, 0> >; auto d = 499; auto i =
 1; auto j = 0; auto V0 = meta_programming::type_map<S_do_compose*>; auto ...Vs = {4, square<int>,
 5, square<int>, meta_programming::multiply_by<int, 2>, meta_programming::add_by<int, 1>}; auto ..
.Ws = {}; Heaps = {void (*)(meta_programming::auto_pack<>*)}]’
source/1_machine_declarations.hpp:587:54:   required from ‘constexpr auto machine_space::machine_s
tart() [with n = machine_space::PD; auto c = machine_space::f_array<unsigned char (*)(unsigned cha
r), machine_space::f_array<unsigned char, 3>, machine_space::f_array<unsigned char, 3, 31, 1, 0>,
machine_space::f_array<unsigned char, 2, 18, 0>, machine_space::f_array<unsigned char, 3, 14, 0, 0
> >; auto d = 500; auto i = 0; auto j = 0; auto ...Vs = {meta_programming::type_map<S_do_compose*>
, 4, square<int>, 5, square<int>, meta_programming::multiply_by<int, 2>, meta_programming::add_by<
```

Debugging

- It doesn’t get much better with other compilers either:

  (clang)

Debugging

```
./source/1_machine_declarations.hpp:587:32: note: in instantiation of function template specializa
tion 'machine_space::machine<'\x1F', '\x01'>::result<machine_space::PD, &machine_space::f_array, 4
99, '\x01', '\x00', &meta_programming::type_map, 4, &square, 5, &square, &meta_programming::multip
ly_by, &meta_programming::add_by, void (*)(meta_programming::auto_pack<> *)>' requested here
                return machine_trampoline<d>(MACHINE(n, c, d, i, j)(U_pack_Vs<>, U_pack_Vs<>));
                                             ^
./source/define_machine_macros.hpp:56:15: note: expanded from macro 'MACHINE'
                >::template result
                      \
                            ^
./case-studies/2_filter.hpp:202:11: note: in instantiation of function template specialization 'ma
chine_space::machine_start<machine_space::PD, &machine_space::f_array, 500, '\x00', '\x00', &meta_
programming::type_map, 4, &square, 5, &square, &meta_programming::multiply_by, &meta_programming::
add_by>' requested here
                        return machine_start<PD, c, d, i, j, U_do_compose, length-1, Vs...>();
                               ^
./case-studies/2_filter.hpp:207:30: note: in instantiation of function template specialization 'f_
do_compose<500, &square, 5, &square, &meta_programming::multiply_by, &meta_programming::add_by>' r
equested here
        constexpr auto do_compose = f_do_compose<500, Vs...>();
                                    ^
main.cpp:238:26: note: in instantiation of variable template specialization 'do_compose' requested
 here
```

Debugging

How do we mitigate this?

- At the machine level we can use basic tools like static_assert.
- At the controller (assembly language) level, it actually becomes much more manageable.
- To give an example as to why, let’s now look at some code from an actual program written with our register machine language:

```
constexpr auto r_filter_contr = r_controller
<
        r_label // is loop end:
        <
                test       < eq          , n   , c_0       >,
                branch     < return_pack                   >,
                apply      < n           , sub , n   , c_1 >,
                check      < cond        , val             >,
                branch     < pop_value                     >,
                rotate_sn  < val                           >,
                goto_contr < is_loop_end                   >

        >, r_label // pop value:
        <
                erase      < val                           >,
                goto_contr < is_loop_end                   >

        >, r_label // return pack:
        <
                pop        < six                           >,
                pack       <                               >
        >
>;
```

Debugging

- Debugging becomes reasonable at this level because we now have access to a long history of well known debugging techniques.

- The first of which is the “naive” approach where we add in a temporary “print” instruction and step through each line, rerunning (i.e. recompiling) until the program breaks:

Debugging

```
r_label // is loop end:
<
        test        < eq          , n   , c_0       >,
        branch      < return_pack                   >,
        apply       < n           , sub , n   , c_1 >,
                                                      // halts,
        stop        < val                           >, // returning
                                                      // val.
        check       < cond        , val             >,
        branch      < pop_value                     >,
        rotate_sn   < val                           >,
        goto_contr  < is_loop_end                   >
>
```

Debugging

```
r_label // is loop end:
<
        test        < eq           , n   , c_0      >,
        branch      < return_pack                   >,
        apply       < n            , sub , n   , c_1 >,
        check       < cond         , val            >,
        branch      < pop_value                     >,
                                                       // halts,
        stop        < val                           >, // returning
                                                       // val.
        rotate_sn   < val                           >,
        goto_contr  < is_loop_end                   >
>
```

Debugging

- Beyond the naive approach, we could also write compile time debuggers to go through these controller style programs and report bugs the way real compilers would.

- It should be noted that I don’t currently have any such tools implemented for my library, but I’m open to adding them in later versions if people show interest.

Debugging

- Lastly, there is an additional reason the designs presented so far are effective at mitigating bugs: They share a common form across machines.

- Because of this, C Macros can be used to simplify the source code redundancy: They reduce the chance of accidentally introducing typos or semantic bugs which might not otherwise be caught by the compiler.

7. Performance

Performance

- I think the question many would like to know at this point is whether or not this design is performant?

- I claim it is, but I have only recently finished the release candidate for version 1.0 of my library, and unfortunately I do not have any benchmarks just yet.
- I make this claim because I have adhered to several key optimization paradigms throughout:

Performance

The block optimization paradigm:

- This innovation comes from Odin Holmes who talks about it on his blog. He uses the name *fast tracking*.
- The idea of blocking is to perform calculations on variadic packs in powers of two (blocks), rather than just one at a time:
- Eg. { V0, V1, V2, V3, Vs... } instead of { V0, Vs... }
- These make up the block module of the machine hierarchy.

Performance

- This optimization works by turning a linear algorithm into a logarithmic one, creating orders of magnitude speed ups.

- It should be noted that the savings don’t come from the block processing itself: Every time info is passed from one machine to the next, that info is copied. Blocking works largely because we are copying exponentially less info.
- Also note that blocking performs an auxiliary optimization in that it saves on nesting depths.

Performance

- Mutator optimization:

- A complete register machine requires many individual instructions to make it work, but there’s still a core of mutator instructions (such as erase, insert, replace) that in general are used more often than the rest.
- In this design, these mutators have both their general purpose versions, but they also have versions optimized for the first eight registers.

Performance

- Machine call optimization:

- Since each machine shares the same name, calling the *next* machine is achieved either through overload resolution or class specialization, or a combination of both.
- Having tested these variations, I have found the hybrid approach of using class specialization (for the names and notes) and function overloading (for the registers) achieves orders of magnitude increases for our machine calls.

Performance

- Dispatch optimization:

- The controller dispatchers serve not only to move from one machine to the next, they also act as optimizers:

  i. They help mitigate the nesting depth problem by requiring less calls.

  ii. Less machine calls also means less stack copying.

Performance

- Finally, although it’s only currently anecdotal, I will add that the small programs I have built generally run in the range of 1-2 seconds for small and medium input and aren’t much slower than the more direct constexpr implementations.

- As this might not be entirely convincing, it is finally time for some demonstrations.

Demonstrations

1. The Factorial Function
2. The Fibonacci Function
3. Filtering and Function Composition

Closing

Caveats

- Undefined Behaviour?

- Open questions:
  i. How many heaps should we include?
  ii. Which supplementary machines to add to the library?

References:

- https://en.wikibooks.org/wiki/C%2B%2B_Programming/Templates/Template_Meta-Programming#History_of_TMP
- https://github.com/hanickadot/compile-time-regular-expressions
- J.E. Hopcroft, R. Motwani, J.D. Ullman. Introduction to Automata Theory, Languages, and Computation (second edition). Addison-Wesley Publishing (2001).
- E. Riehl. Category Theory in Context. Dover Publications, Inc. (2016).
- https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html