C++ is a Metacompiler

Daniel Nikpayuk

2024

Outline

Outline

Practical:

Outline

Practical:

1. Introduction
2. Motivation
3. Demonstration
4. Performance

Outline

Practical:

1. Introduction
2. Motivation
3. Demonstration
4. Performance

Theory:

Outline

Practical:

1. Introduction
2. Motivation
3. Demonstration
4. Performance

Theory:

5. Philosophy
6. Methodology
7. Library
8. Entailment

1st half:

Practical

Introduction

Who am I?

- I'm a self-taught coder.
- I've been programming in C++ since 2005.
- I don't currently work in the tech industry.
- I have a Bachelor of Arts majoring in mathematics, minoring in economics.
- I am an Inuit person (specifically Inuvialuit) from Canada's western Arctic.
- I am devoted to the continued renewal of my people's language and culture.

Why C++?

- It is a life goal of mine to build a programming language for multimedia production.
- I hope to offer said language as an option for telling and retelling my people's stories, traditional and new.

Figure: inuksuk

- Such a language will generally require systems level performance, and so C++ is a good fit for writing its first compiler.

Motivation

What is a metacompiler?

What is a metacompiler? The short answer for now. . .

What is a metacompiler? The short answer for now...
is it's a compile time compiler.

What is a metacompiler? The short answer for now. . .
is it's a compile time compiler.

My talk is about the working metacompiler that I have built, and this section provides my motivation for doing so.

How long have I been working on this project?

How long have I been working on this project?
Directly, about 2 years.

How long have I been working on this project?
Directly, about 2 years. Indirectly,

How long have I been working on this project?
Directly, about 2 years. Indirectly, about 10 years.

What would motivate me

What would motivate me
to work on this for 10 years?

What would motivate me
to work on this for 10 years?

loops.

When I first starting programming,

When I first starting programming, the initial years were about getting to know the language.

After that,

After that,

I spent time writing in the Qt framework,

After that,

I spent time writing in the Qt framework,
because graphical programming is fun.

I had some success,
I even wrote some (unpublished) smartphone apps,

I had some success,
I even wrote some (unpublished) smartphone apps,

but after a while I plateaued and decided it was time to improve my skillset.

I had some success,
I even wrote some (unpublished) smartphone apps,

but after a while I plateaued and decided it was time to improve my skillset.

What did I do?

I reproduced the standard library containers, such as vector.

I reproduced the standard library containers, such as vector.

In doing so, I found myself writing many loops which I felt were both small variations of each other,

I reproduced the standard library containers,
such as vector.

In doing so, I found myself writing many loops which
I felt were both small variations of each other,
and tedious to write.

```
while (in != end)
{
    perform_action(in);
    increment(in);
}
```

```
while (in <= end)
{
    perform_action(in);
    increment(in);
}
```

```
while (in != end)
{
    perform_action(in);
    increment(in);
}

perform_action(in);
```

```
increment(in);

while (in != end)
{
   perform_action(in);
   increment(in);
}

perform_action(in);
```

With loops like these,

With loops like these,
I was spending more effort thinking about
the syntax

With loops like these,
I was spending more effort thinking about
the syntax than the semantics of the loop.

The tradeoff with this syntax comes from the idea of scope.

The tradeoff with this syntax
comes from the idea of scope.

With this style of grammar, scope is hardcoded.

The tradeoff with this syntax
comes from the idea of scope.

With this style of grammar, scope is hardcoded.
This limits our options when it comes to loops.

The tradeoff with this syntax
comes from the idea of scope.

With this style of grammar, scope is hardcoded.
This limits our options when it comes to loops.

Scope is good though:

The tradeoff with this syntax
comes from the idea of scope.

With this style of grammar, scope is hardcoded.
This limits our options when it comes to loops.

Scope is good though: If you've ever written assembly,

The tradeoff with this syntax
comes from the idea of scope.

With this style of grammar, scope is hardcoded.
This limits our options when it comes to loops.

Scope is good though: If you've ever written assembly,
you realize the importance of scope.

The tradeoff with this syntax
comes from the idea of scope.

With this style of grammar, scope is hardcoded.
This limits our options when it comes to loops.

Scope is good though: If you've ever written assembly,
you realize the importance of scope.

So I had to find another way.

Ranges are the standard solution to this problem, Ranges are the standard solution to this problem,

I designed an alternative called: Intervals.

I borrow the interval notation from math.

I borrow the interval notation from math.

The idea of an interval is it encodes an iteration, but with options which communicate how the iteration algorithm acts on its endpoints.

I borrow the interval notation from math.

The idea of an interval is it encodes an iteration, but with options which communicate how the iteration algorithm acts on its endpoints.

The options are defined as follows:

- [ ] - closed
- ( ] - opening
- ( ) - open
- [ ) - closing

- [ ] - `closed`: acts on the left and right endpoints.
- ( ] - `opening`
- ( ) - `open`
- [ ) - `closing`

- [ ] - `closed`
- ( ] - `opening`: skips the left, acts on the right.
- ( ) - `open`
- [ ) - `closing`

- [ ] - `closed`
- ( ] - `opening`
- ( ) - `open`: skips the left, doesn't act on the right.
- [ ) - `closing`

- [ ] - `closed`
- ( ] - `opening`
- ( ) - `open`
- [ ) - `closing`: acts on the left, not on the right.

```
// closed:

sum [1, 3] == 6
```

```
// opening:

sum (1, 3] == 5
```

```
// open:

sum (1, 3) == 2
```

```
// closing:

sum [1, 3) == 3
```

As for my motivation:

As for my motivation:

To summarize,

As for my motivation:

To summarize,
I wanted access to a domain specific language (DSL),

As for my motivation:

To summarize,
I wanted access to a domain specific language (DSL),
within C++ itself.

Demonstration

What is a metacompiler?

What is a metacompiler?

It's a compile time compiler,

What is a metacompiler?

It's a compile time compiler, which compiles DSLs.

In this section I introduce my chord DSL.

In this section I introduce my chord DSL.

It's not the point of this talk,

In this section I introduce my chord DSL.

It's not the point of this talk,
but it does demonstrate what's possible

In this section I introduce my chord DSL.

It's not the point of this talk,
but it does demonstrate what's possible
given a working metacompiler.

example:

square

```
main x                ;

body:                 ;
  . = multiply x x    ;
  return _            ;
```

```
main x                  ;

 body:                  ;
   . = multiply x x     ;
   return _             ;
```

```
main x                  ;

body:                   ;
  . = multiply x x      ;
  return _              ;
```

```
main x                 ;

body:                  ;
  . = multiply x x ;
  return _             ;
```

```
main x                ;

body:                 ;
  . = multiply x x    ;
  return _            ;
```

```
main x                 ;

body:                  ;
  . = multiply x x     ;
  return _             ;
```

```
main x                  ;

body:                   ;
  . = multiply x x      ;
  return _              ;
```

```
main x                  ;

body:                   ;
  . = multiply x x      ;
  return _              ;
```

example:
sum of squares

```
main x y                    ;

body:                       ;
  x = multiply x x ;
  y = multiply y y ;
  . = add        x y ;
  return _                  ;
```

example:

$$(x+1)^2$$

```
main x                   ;

body:                    ;
  . = add x 1            ;
  . = multiply _ _       ;
  return _               ;
```

```
main x                    ;

body:                     ;
  . = add x 1             ;
  . = multiply _ _        ;
  return _                ;
```

example:

exponents

```
main t                          ;

body:                           ;
  . = multiply t t t t t        ;
  return _                      ;
```

example:
square (argpose)

```
main x                               ;

vars:                                ;
  declare sq                         ;
defs:                                ;
  sq # argpose[1]{multiply 0 0}      ;
body:                                ;
  . = sq x                           ;
  return _                           ;
```

```
main x                                 ;

vars:                                  ;
  declare sq                           ;
defs:                                  ;
  sq # argpose[1]{multiply 0 0}        ;
body:                                  ;
  . = sq x                             ;
  return _                             ;
```

```
main x                                  ;

vars:                                   ;
  declare sq                            ;
defs:                                   ;
  sq # argpose[1]{multiply 0 0}         ;
body:                                   ;
  . = sq x                              ;
  return _                              ;
```

```
main x                                  ;

vars:                                   ;
  declare sq                            ;
defs:                                   ;
  sq # argpose[1]{multiply 0 0}         ;
body:                                   ;
  . = sq x                              ;
  return _                              ;
```

```
main x                                  ;

vars:                                   ;
  declare sq                            ;
defs:                                   ;
  sq # argpose[1]{multiply 0 0}         ;
body:                                   ;
  . = sq x                              ;
  return _                              ;
```

```
main x                                        ;

vars:                                         ;
  declare sq                                  ;
defs:                                         ;
  sq # argpose[1]{multiply 0 0}               ;
body:                                         ;
  . = sq x                                    ;
  return _                                    ;
```

example:
sum of squares (subpose)

```
main x y                                        ;

vars:                                           ;
  declare sq sum_of_sq                          ;

defs:                                           ;
  sq        # argpose[1]{multiply 0 0}          ;
  sum_of_sq # subpose[2]{add sq sq}             ;

body:                                           ;
  . = sum_of_sq x y                             ;
  return _                                      ;
```

```
main x y                                           ;

vars:                                              ;
  declare sq sum_of_sq                             ;

defs:                                              ;
  sq        # argpose[1]{multiply 0 0}             ;
  sum_of_sq # subpose[2]{add sq sq}                ;

body:                                              ;
  . = sum_of_sq x y                                ;
  return _                                         ;
```

```
main x y                                              ;

vars:                                                 ;
  declare sq sum_of_sq                                ;

defs:                                                 ;
  sq        # argpose[1]{multiply 0 0}                ;
  sum_of_sq # subpose[2]{add sq sq}                   ;

body:                                                 ;
  . = sum_of_sq x y                                   ;
  return _                                            ;
```

```
main x y                                          ;

vars:                                             ;
  declare sq sum_of_sq                            ;

defs:                                             ;
  sq        # argpose[1]{multiply 0 0}            ;
  sum_of_sq # subpose[2]{add sq sq}               ;

body:                                             ;
  . = sum_of_sq x y                               ;
  return _                                        ;
```

```
main x y                                        ;

vars:                                           ;
  declare sq sum_of_sq                          ;

defs:                                           ;
  sq        # argpose[1]{multiply 0 0}          ;
  sum_of_sq # subpose[2]{add sq sq}             ;

body:                                           ;
  . = sum_of_sq x y                             ;
  return _                                      ;
```

```
main x y                                       ;

vars:                                          ;
  declare sq sum_of_sq                         ;

defs:                                          ;
  sq         # argpose[1]{multiply 0 0}        ;
  sum_of_sq # subpose[2]{add sq sq}            ;

body:                                          ;
  . = sum_of_sq x y                            ;
  return _                                     ;
```

example:

twice (curry)

```
main x                                    ;

vars:                                     ;
  declare twice                           ;
defs:                                     ;
  twice # curry[1]{multiply two}          ;
body:                                     ;
  . = twice x                             ;
  return _                                ;


, binding("two", 2)
```

```
main x                                 ;

vars:                                  ;
  declare twice                        ;
defs:                                  ;
  twice # curry[1]{multiply two}       ;
body:                                  ;
  . = twice x                          ;
  return _                             ;


, binding("two", 2)
```

```
main x                                      ;

vars:                                       ;
  declare twice                             ;
defs:                                       ;
  twice # curry[1]{multiply two}            ;
body:                                       ;
  . = twice x                               ;
  return _                                  ;


, binding("two", 2)
```

```
main x                                   ;

vars:                                    ;
  declare twice                          ;
defs:                                    ;
  twice # curry[1]{multiply two}         ;
body:                                    ;
  . = twice x                            ;
  return _                               ;


, binding("two", 2)
```

```
main x                                    ;

vars:                                     ;
  declare twice                           ;
defs:                                     ;
  twice # curry[1]{multiply two}          ;
body:                                     ;
  . = twice x                             ;
  return _                                ;
```

```
, binding("two", 2)
```

```
main x                                      ;

vars:                                       ;
  declare twice                             ;
defs:                                       ;
  twice # curry[1]{multiply two}            ;
body:                                       ;
  . = twice x                               ;
  return _                                  ;


, binding("two", 2)
```

example:

factorial (naive)

```
type T                ;
factorial n -> T      ;

body:                 ;
  test equal n 0      ;
  branch done         ;
  . = subtract n 1    ;
  . = factorial _     ;
  . = multiply n _    ;
  return _            ;

done:                 ;
  return 1:T          ;
```

```
type T                  ;
factorial n -> T        ;
```

```
type T                ;
factorial n -> T      ;
```

```
type T                  ;
factorial n -> T        ;
```

```
type T                  ;
factorial n -> T        ;
```

```
body:                 ;
  test equal n 0      ;
  branch done         ;
  . = subtract n 1    ;
  . = factorial _     ;
  . = multiply n _    ;
  return _            ;
```

```
done:                   ;
  return 1:T            ;
```

```
done:                ;
  return 1:T         ;
```

example: factorial (goto)

```
main p n              ;

loop:                 ;
  test is_zero n      ;
  branch done         ;
  p = multiply p n    ;
  n = decrement n     ;
  goto loop           ;

done:                 ;
  return p            ;
```

Why is this language called chord?

Why is this language called chord?

Chord is a high level assembly language, which provides direct support for the classical functional operators:

Why is this language called chord?

Chord is a high level assembly language, which provides direct support for the classical functional operators:

$$\{\text{repeat}, \text{map}, \text{fold}, \text{find}, \text{sift}\}$$

In particular,

In particular,
I modified these operators to work well with C++ iterators.

In particular,
I modified these operators to work well with C++ iterators.

I wanted a distinct terminology to describe these new grammatical patterns,

In particular,
I modified these operators to work well with C++ iterators.

I wanted a distinct terminology to describe these new grammatical patterns, and so I loosely borrowed from music terminology.

I'm not a musician, so please be kind:

I'm not a musician, so please be kind:

- I first rename an iterator to be a note.

I'm not a musician, so please be kind:

- I first rename an iterator to be a note.
- Two notes together define an interval: $[\text{in}, \text{end})$

I'm not a musician, so please be kind:

- I first rename an iterator to be a note.
- Two notes together define an interval: $[\text{in}, \text{end})$
- A sequence of intervals is a chord: $[\,,\,)\ [\,)\ [\,]$

I'm not a musician, so please be kind:

- I first rename an iterator to be a note.
- Two notes together define an interval: $[\text{in}, \text{end})$
- A sequence of intervals is a chord: $[\,,\,)\ [\,)\ [\,]$
- A sequence of chords is a progression.

Refering back to my original motivation:

Refering back to my original motivation:

I wanted to write loops with a grammar

Refering back to my original motivation:

I wanted to write loops with a grammar that let me focus on the semantics of the algorithm, not the syntax.

This started with the idea of an interval:

```
// closed:

sum [1, 3] == 6
```

example:

sum (fold)

```
main out in end                           ;

vars:                                     ;
  declare sum                             ;
defs:                                     ;
  sum # fold[1]{add * @|||} <> [,]        ;
body:                                     ;
  . = sum !out in end                     ;
  return _                                ;
```

```
fold[1]{add * @|||} <> [,]
```

`fold[N]{combine|act|mutate|break}`

`fold[N]{combine|act|mutate|break}`

```
fold[N]{combine|act|mutate|break}
```

`fold[N]{combine|act|mutate|break}`

`fold[N]{combine|act|mutate|break}`

```
fold[N]{combine|act|mutate|break}
```

```
fold[1]{add * @|||} <> [,]
```

```
defs:                                      ;

  sum # fold[1]{add * @|||} <> [,] ;

body:                                      ;
  . = sum !out in end                      ;
  return _                                 ;
```

```
defs:                                   ;

  sum # fold[1]{add * @|||} <> [,]      ;

body:                                   ;
  . = sum !out in end                   ;
  return _                              ;
```

```
defs:                                    ;

  sum # fold[1]{add * @|||} <> [,] ;

body:                                    ;
  . = sum !out in end                    ;
  return _                               ;
```

```
defs:                                   ;

  sum # fold[1]{add * @|||} <> [,] ;

body:                                   ;
  . = sum !out in end                   ;
  return _                              ;
```

```
defs:                                          ;

  sum # fold[1]{add * @|||} <> [,] ;

body:                                          ;
  . = sum !out in end                          ;
  return _                                     ;
```

```
defs:                                   ;

  sum # fold[1]{add * @|||} <> [,]      ;

body:                                   ;
  . = sum !out in end                   ;
  return _                              ;
```

```
defs:                                    ;

  sum # fold[1]{add * @|||} <> [,] ;

body:                                    ;
  . = sum !out in end                    ;
  return _                               ;
```

example:
vector addition (map)

```
main out in end in1                        ;

vars:                                      ;
  declare vec_add                          ;
defs:                                      ;
  vec_add # map[2]{add||} [) [,) [)        ;
body:                                      ;
  . = vec_add !out in end in1              ;
  return _                                 ;
```

```
defs:                                      ;

  vec_add # map[2]{add||} [) [,) [)        ;

body:                                      ;
  . = vec_add !out in end in1              ;
  return _                                 ;
```

example:
change of base (progression)

$$12_{\,(\text{base } 10)} \quad \longrightarrow \quad ?_{\,(\text{base } 2)}$$

$$12_{\,(\text{base } 10)} \quad \rightarrow \quad ?_{\,(\text{base } 2)}$$

$$12 \quad \rightarrow \quad (\,12,\ 6,\ 3,\ 1,\ 0\,)$$

$$12_{\,(\text{base } 10)} \quad \rightarrow \quad ?_{\,(\text{base } 2)}$$

$$12 \quad \rightarrow \quad (\,12,\ 6,\ 3,\ 1,\ 0\,)$$

$$\rightarrow \quad (\ \ 0,\ 0,\ 1,\ 1,\ 0\,)$$

$$12_{\,(\text{base } 10)} \quad \rightarrow \quad ?_{\,(\text{base } 2)}$$

$$12 \quad \rightarrow \quad (\,12,\ 6,\ 3,\ 1,\ 0\,)$$

$$\rightarrow \quad (\ \ 0,\ 0,\ 1,\ 1,\ 0\,)$$

$$\rightarrow \quad 01100 \quad (\text{reverse})$$

```
main out in                                                  ;

vars:                                                        ;
  declare change_base print_change                           ;
defs:                                                        ;
  change_base  # map[1]{rem_by_n @||} [) [div_by_n|~,)       ;
  print_change # repeat[1]{|print * @|} (-|+,] <>            ;
body:                                                        ;
  . = change_base !out in 0                                  ;
  . = print_change _ out format                              ;
  return _                                                   ;


            // template<auto radix>
            , binding( "print"    , _print_              )
            , binding( "div_by_n" , _divide_by_<radix>   )
            , binding( "rem_by_n" , _modulo_by_<radix>   )
            , binding( "format"   , strlit_type{"%d"}    )
```

```
// template<auto radix>
, binding( "print"     , _print_               )
, binding( "div_by_n"  , _divide_by_<radix>    )
, binding( "rem_by_n"  , _modulo_by_<radix>    )
, binding( "format"    , strlit_type{"%d"}     )
```

```
defs:                                                        ;
 change_base  # map[1]{rem_by_n @||} [) [div_by_n|~,)        ;
 print_change # repeat[1]{|print * @|} (-|+,] <>             ;
body:                                                        ;
 . = change_base !out in 0                                   ;
 . = print_change _ out format                               ;
 return _                                                    ;
```

```
map[1]{rem_by_n @||} [) [div_by_n|~,) ;
```

```
map[1]{rem_by_n @||} [) [div_by_n|~,) ;
```

```
map[1]{rem_by_n @||} [) [div_by_n|~,) ;
```

```
map[1]{rem_by_n @||} [) [div_by_n|~,) ;
```

```
repeat[1]{|print * @|} (-|+,] <> ;
```

```
repeat[1]{|print * @|} (-|+,] <> ;
```

```
repeat[1]{|print * @|} (-|+,] <> ;
```

```
repeat[1]{|print * @|} (-|+,] <> ;
```

```
repeat[1]{|print * @|} (-|+,] <> ;
```

```
defs:                                                    ;
 change_base  # map[1]{rem_by_n @||} [) [div_by_n|~,)   ;
 print_change # repeat[1]{|print * @|} (-|+,] <>        ;
body:                                                    ;
 . = change_base !out in 0                              ;
 . = print_change _ out format                          ;
 return _                                               ;
```

```
body:                               ;
  . = change_base !out in 0         ;
  . = print_change _ out format     ;
  return _                          ;
```

```
body:                               ;
  . = change_base !out in 0         ;
  . = print_change _ out format     ;
  return _                          ;
```

Hustle

What is a metacompiler?

What is a metacompiler?

It's a compile time compiler,

What is a metacompiler?

It's a compile time compiler, which compiles DSLs.

What is a metacompiler?

It's a compile time compiler, which compiles DSLs.
The key phrase being DSLs.

Here I also introduce my hustle DSL.

Here I also introduce my hustle DSL.

It is as close to the scheme (lisp) programming language that this paradigm will allow.

example:

factorial

```
(type T
  (define (factorial n) -> T
    (if (= n 0)
      1:T
      (* n (factorial (- n 1)))
    )))
```

Performance

This short section is informal.

This short section is informal.

Before we commit to detailed theory (in the following),

This short section is informal.

Before we commit to detailed theory (in the following), I thought I'd offer some basic stats here.

First, a reality check.

In the previous section we introduced the hustle DSL:

```
(type T
  (define (factorial n) -> T
    (if (= n 0)
      1:T
      (* n (factorial (- n 1)))
    )))
```

In C++17,

In order to metacompile it,
we actually write it like this:

```
constexpr auto _hustle_factorial_v0()
{
  return source
  (
     "(type T                          "
     "  (define (factorial n) -> T     "
     "    (if (= n 0)                  "
     "      1:T                        "
     "      (* n (factorial (- n 1))) "
     "    )                            "
     "  )                              "
     ")                                "
  );
}
```

It's a string literal we wrap in a function, It's a string literal we wrap in a function, so we can pass it as a template parameter:

```
using factorial_v0 = hustle::metacompile
<
        _hustle_factorial_v0,
        null_env, unsigned long
>;

static_assert(factorial_v0::result(9) == 362880);
```

Our metacompiler is a compile time compiler,

Our metacompiler is a compile time compiler,

That turns C++ string literals into C++ constexpr functions.

Our metacompiler is a compile time compiler,

That turns C++ string literals into C++ constexpr functions. They can be used both at compile time and at run time.

We won't go into detail here,

We won't go into detail here,

but a quick explanation as to how this works

We won't go into detail here,

but a quick explanation as to how this works is that a metacompiler turns this:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                 ;"
                "factorial n -> T       ;"

                "body:                  ;"
                "   test equal n 0      ;"
                "   branch done         ;"
                "    . = subtract n 1   ;"
                "    . = factorial _    ;"
                "    . = multiply n _   ;"
                "   return _            ;"

                "done:                  ;"
                "   return 1:T          ;"
        );
}
```

into this:

```
constexpr size_type value[][8] =
{
    { MN::id          , MT::id        ,  0,  0,  0,  0,  0,  1 },
    { MN::hash        , MT::port      ,  5,  0,  0,  0,  0,  1 },
    { MN::pad         , MT::select    ,  0,  1,  0,  0,  0,  1 },
    { MN::pad         , MT::id        ,  0,  0,  0,  0,  0,  1 },
    { MN::go_to       , MT::id        , 50,  0,  0,  0,  0,  1 },
    { MN::id          , MT::id        ,  0,  0,  0,  0,  0,  1 },
    { MN::eval        , MT::back      ,  7,  0,  0,  0,  0,  4 },
    { MN::id          , MT::id        ,  0,  0,  0,  0,  0,  1 },
    { MN::lookup      , MT::first     ,  0,  7,  0,  0,  0,  1 },
    { MN::halt        , MT::first     ,  0,  0,  0,  0,  0,  1 },
    { MN::eval        , MT::back      , 11,  0,  0,  0,  0,  5 },
    { MN::id          , MT::id        ,  0,  0,  0,  0,  0,  1 },
    { MN::arg         , MT::select    ,  1,  0,  0,  0,  0,  1 },
    { MN::arg         , MT::drop      ,  0,  0,  0,  0,  0,  1 },
    { MN::halt        , MT::first     ,  0,  0,  0,  0,  0,  1 },
    { MN::type        , MT::n_number  ,  0,  0,  0,  0,  0,  1 },
    { MN::literal     , MT::back      ,  0,  0,  0,  0,  0,  1 },
```

which we then pass to this:

```cpp
template<auto... filler>
struct T_machine<MN::hash, MT::id, filler...>
{
    template<NIK_MACHINE_PARAMS(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
            constexpr auto ni = MD<c>::pos(i);
            constexpr auto nv = U_machine_compound<c, ni>;

            return NIK_MACHINE_TEMPLATE(c, i)
            ::NIK_MACHINE_RESULT_2TS(c, i, l, t, r, decltype(nv), Ts...)
                    (nv, vs...);
    }
};
```

which finally turns into a constexpr function.

which finally turns into a constexpr function.

As for performance. . .

I present to you a non-trivial algorithm:

I present to you a non-trivial algorithm:

The square root function,

I present to you a non-trivial algorithm:

The square root function,
implemented using newton's method,

I present to you a non-trivial algorithm:

The square root function,
implemented using newton's method,
written in the hustle DSL:

```
(type T
  (define (sqrt x)

    (define (square y) (* y y))
    (define (abs y) (if (< y 0) (- y) y))
    (define (good-enough? guess) (< (abs (- (square guess) x)) tolerance))
    (define (average y z) (/ (+ y z) 2))
    (define (improve guess) (average guess (/ x guess)) )
    (define (sqrt-iter guess) -> T
      (if (good-enough? guess) guess (sqrt-iter (improve guess)))
    )

    (sqrt-iter 1:T)))

, binding("tolerance", 0.0001)
```

Before showing performance stats on this algorithm, Before showing performance stats on this algorithm, let's observe the stats if it were reimplemented in standard C++17:

```
template<typename T>
constexpr auto sqrt_iter(T x, T guess) -> T
{
    auto tolerance   = 0.0001;
    auto square      = [ ](T y){ return y * y; };
    auto abs         = [ ](T y){ return (y < 0) ? -y : y; };
    auto good_enough = [&](T g){ return (abs(square(g) - x) < tolerance); };

    auto average     = [ ](T y, T z){ return (y + z) / 2; };
    auto improve     = [&](T g){ return average(g, x/g); };

    if (good_enough(guess)) return guess;
    else                    return sqrt_iter(x, improve(guess));
}

template<typename T>
constexpr auto sqrt(T x) { return sqrt_iter<T>(x, 1.0); }
```

Perf (Linux, GCC 11.4.0, Clang 14.0.0):

Perf (Linux, GCC 11.4.0, Clang 14.0.0):

	gcc	clang
compile time	0.100 s	0.114 s
run time	0.001 s	0.001 s
-01 binary size	16 176 B	16 104 B
-02/03 binary size	16 128 B	16 104 B

```
(type T
  (define (sqrt x)

    (define (square y) (* y y))
    (define (abs y) (if (< y 0) (- y) y))
    (define (good-enough? guess) (< (abs (- (square guess) x)) tolerance))
    (define (average y z) (/ (+ y z) 2))
    (define (improve guess) (average guess (/ x guess)) )
    (define (sqrt-iter guess) -> T
      (if (good-enough? guess) guess (sqrt-iter (improve guess)))
    )

    (sqrt-iter 1:T)))

, binding("tolerance", 0.0001)
```

Perf (Linux, GCC 11.4.0, Clang 14.0.0):

Perf (Linux, GCC 11.4.0, Clang 14.0.0):

	gcc	clang
compile time	0.390 s	0.842 s
run time	0.002 s	0.001 s
-01 binary size	17 432 B	16 624 B
-02/03 binary size	16 272 B	16 632 B

2nd half:

Theory

This half of the talk is much more academic.

This half of the talk is much more academic.

Be warned:

This half of the talk is much more academic.
Be warned:

I introduce a lot of my own terminology.

This half of the talk is much more academic.
Be warned:

I introduce a lot of my own terminology.
Reasoning about a metacompiler involves a lot of context switching,

This half of the talk is much more academic. Be warned:

I introduce a lot of my own terminology. Reasoning about a metacompiler involves a lot of context switching, and so having distinct terms helps keep contextual boundaries clear.

Philosophy

Before we can engineer, we need to design.

Before we can engineer, we need to design.
A good start is to question our terms.

Before we can engineer, we need to design.
A good start is to question our terms.

This section is about design,

Before we can engineer, we need to design.
A good start is to question our terms.

This section is about design, and the philosophy behind the metacompiler paradigm.

To begin:

To begin:

- A metacompiler is a compile time compiler.

To begin:

- A metacompiler is a compile time compiler.
- C++ is a metacompiler.

To begin:

- A metacompiler is a compile time compiler.
- C++ is a metacompiler.
- We've seen what a metacompiler *does*.

To begin:

- A metacompiler is a compile time compiler.
- C++ is a metacompiler.
- We've seen what a metacompiler *does*.
- We now ask what a metacompiler *is*.

Let's take a short tour
of related concepts.

We ask the following questions:

We ask the following questions:

- What is a compiler?

We ask the following questions:

- What is a compiler?
- What is an interpreter?

We ask the following questions:

- What is a compiler?
- What is an interpreter?
- What is a transpiler?

To keep things simple:

To keep things simple:

- A *compiler* takes source code and translates it into assembly.

To keep things simple:

- A *compiler* takes source code and translates it into assembly.
- An *interpreter* takes source code, translates, then executes it directly.

To keep things simple:

- A *compiler* takes source code and translates it into assembly.
- An *interpreter* takes source code, translates, then executes it directly.
- A *transpiler* takes source code and translates it into the source code of another language.

Do these ideas apply to a metacompiler?

Do these ideas apply to a metacompiler?

- Compiler: Yes, a metacompiler takes source code and translates it into an intermediate assembly.

Do these ideas apply to a metacompiler?

- Compiler: Yes, a metacompiler takes source code and translates it into an intermediate assembly.
- Interpreter: Maybe, a metacompiled function can be executed at compile time.

Do these ideas apply to a metacompiler?

- Compiler: Yes, a metacompiler takes source code and translates it into an intermediate assembly.
- Interpreter: Maybe, a metacompiled function can be executed at compile time.
- Transpiler: Maybe, a metacompiler takes source code and does translate it into C++, but only C++.

So I've called it a metacompiler.

So I've called it a metacompiler.

What makes it "meta?"

So I've called it a metacompiler.

What makes it "meta?"

- The prefix comes from metaprogramming.

So I've called it a metacompiler.

What makes it "meta?"

- The prefix comes from metaprogramming.
- In C++ this means compile time programming.

So I've called it a metacompiler.

What makes it "meta?"

- The prefix comes from metaprogramming.
- In C++ this means compile time programming.
- This includes constexpr time programming,

So I've called it a metacompiler.

What makes it "meta?"

- The prefix comes from metaprogramming.
- In C++ this means compile time programming.
- This includes constexpr time programming, as well as template metaprogramming.

As such, a metacompiler requires
we refine our notion of time.

We ask:

We ask:

- What is a timescape?

We ask:

- What is a timescape?
- What is a timescope?

The short answer:

When observing the lifespan of a program, a timescape allows us to decompose it into timescopes.

As for specific timescopes:

As for specific timescopes:

- Run time is when a program is being executed.

As for specific timescopes:

- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.

As for specific timescopes:

- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.
- Metarun time is when a metaprogram is being executed...

As for specific timescopes:

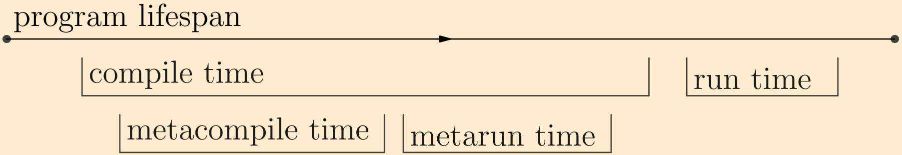
- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.
- Metarun time is when a metaprogram is being executed. . . *within the scope of compile time.*

As for specific timescopes:

- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.
- Metarun time is when a metaprogram is being executed. . . *within the scope of compile time.*
- Metacompile time is when a metaprogram is being translated for execution. . .

As for specific timescopes:

- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.
- Metarun time is when a metaprogram is being executed. . . *within the scope of compile time.*
- Metacompile time is when a metaprogram is being translated for execution. . . *within the scope of compile time.*


program lifespan
compile time
run time
metacompile time
metarun time

What about constexpr time?

What about constexpr time?

- This is C++ specific.

What about constexpr time?

- This is C++ specific.
- This timescope in effect represents either run time or metarun time.

A metacompiler requires we also consider ideas of self similarity.

Why?

Why?

Because in theory we could metacompile source code from the same language that is otherwise being compiled.

Given this, we ask:

Given this, we ask:

- What is a metacircular evaluator?

Given this, we ask:

- What is a metacircular evaluator?
- What is a self-hosting compiler?

Given this, we ask:

- What is a metacircular evaluator?
- What is a self-hosting compiler?
- What is an abstract machine?

Given this, we ask:

- What is a metacircular evaluator?
- What is a self-hosting compiler?
- What is an abstract machine?
- What is a virtual machine?

A metacircular evaluator:

A metacircular evaluator:

- Starts with interpreted language.

A metacircular evaluator:

- Starts with interpreted language.
- Builds a metacircular library.

A metacircular evaluator:

- Starts with interpreted language.
- Builds a metacircular library.
- Builds a function called an evaluator.

A metacircular evaluator:

- Starts with interpreted language.
- Builds a metacircular library.
- Builds a function called an evaluator.
- This evaluator simulates the language's own interpreter.

A metacircular evaluator:

- Starts with interpreted language.
- Builds a metacircular library.
- Builds a function called an evaluator.
- This evaluator simulates the language's own interpreter.
- This evaluator can execute source code from the same language.

A self-hosting compiler:

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

- An interpreter is allowed to interleave source code translation with execution.

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

- An interpreter is allowed to interleave source code translation with execution.
- A compiler is restricted to modularizing source code translation from execution.

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

- An interpreter is allowed to interleave source code translation with execution.
- A compiler is restricted to modularizing source code translation from execution. It *must* translate first, only then can it execute.

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

- An interpreter is allowed to interleave source code translation with execution.
- A compiler is restricted to modularizing source code translation from execution. It *must* translate first, only then can it execute.
- This creates subtle differences in their respective designs.

An abstract machine:

An abstract machine:

- Is a state machine.

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[\,0 \to N\,]$:

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[\,0 \rightarrow N\,]$:

$$S_{0:\text{begin}} \quad \rightarrow \quad S_1 \quad \rightarrow \quad \dots \quad \rightarrow \quad S_{N:\text{end}}$$

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[0 \to N]$:

$$S_{0:\text{begin}} \quad \to \quad S_1 \quad \to \quad \dots \quad \to \quad S_{N:\text{end}}$$

- Such states are usually expected to have the same shape.

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[0 \to N]$:

$$S_{0:\text{begin}} \quad \to \quad S_1 \quad \to \quad \dots \quad \to \quad S_{N:\text{end}}$$

- Such states are usually expected to have the same shape. In effect, you can consider them to be a data structure.

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[0 \rightarrow N]$:

$$S_{0:\text{begin}} \quad \rightarrow \quad S_1 \quad \rightarrow \quad \ldots \quad \rightarrow \quad S_{N:\text{end}}$$

- Such states are usually expected to have the same shape. In effect, you can consider them to be a data structure.
- These machines are generally given some kind of controller (sometimes source code) to direct their computation.

A virtual machine

A virtual machine (in recent literature):

A virtual machine (in recent literature):

- Is an abstract machine.

A virtual machine (in recent literature):

- Is an abstract machine.
- Has states that represent actual hardware.

A virtual machine (in recent literature):

- Is an abstract machine.
- Has states that represent actual hardware.
- Can be used to simulate hardware on top of actual hardware.

Do these ideas apply to a metacompiler?

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.
- self-hosting compiler: Maybe, in theory we could rebuild C++ itself,

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.
- self-hosting compiler: Maybe, in theory we could rebuild C++ itself, but done at compile time it might not be performant enough to be worth it.

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.
- self-hosting compiler: Maybe, in theory we could rebuild C++ itself, but done at compile time it might not be performant enough to be worth it.
- abstract machine: Yes, such machines underly the implementation design.

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.
- self-hosting compiler: Maybe, in theory we could rebuild C++ itself, but done at compile time it might not be performant enough to be worth it.
- abstract machine: Yes, such machines underly the implementation design.
- virtual machine: Somewhat, in theory optimized state transitions can be designed with hardware in mind.

What then is a metacompiler?

What then is a metacompiler?

- It is a toolchain of related technologies which translate source code into assembly.

What then is a metacompiler?

- It is a toolchain of related technologies which translate source code into assembly.
- In terms of the technologies that make up this chain,

What then is a metacompiler?

- It is a toolchain of related technologies which translate source code into assembly.
- In terms of the technologies that make up this chain, it is the idea of a DSL engine that is most relevant to this talk.

What is a DSL engine?

What is a DSL engine?

It is an abstract machine which translates domain specific languages into assembly.

To finish this section we ask one more question.

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly,

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification,

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification, C++17 and later.

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification, C++17 and later.
- C++17 has an emergence of grammar and rules to support a DSL engine,

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification, C++17 and later.
- C++17 has an emergence of grammar and rules to support a DSL engine, one which is also performant.

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification, C++17 and later.
- C++17 has an emergence of grammar and rules to support a DSL engine, one which is also performant.
- It is independent of vendor implementation.

Methodology

A metacompiler as a technology is currently split into the following parts:

A metacompiler as a technology is currently split into the following parts:

- A parser generator.

A metacompiler as a technology is
currently split into the following parts:

- A parser generator.
- DSLs, each having their own lexers, and parsers.

A metacompiler as a technology is currently split into the following parts:

- A parser generator.
- DSLs, each having their own lexers, and parsers.
- A shared intermediate representation (IR).

A metacompiler as a technology is currently split into the following parts:

- A parser generator.
- DSLs, each having their own lexers, and parsers.
- A shared intermediate representation (IR).
- A DSL engine which turns the IR into a constexpr C++ function.

In this section we give a brief history and overview of the theoretical methods needed

In this section we give a brief history and overview of the theoretical methods needed to implement the various components that make up a meta-compiler.

In this section we give a brief history and overview of the theoretical methods needed to implement the various components that make up a meta-compiler.

As for the DSL engine, the actual implementation details will be given in the next section.

Why talk about methods?

Why talk about methods?

A general purpose DSL engine needs to be able to metacompile any language,

Why talk about methods?

A general purpose DSL engine needs to be able to metacompile any language, and so we need its implementation design to be based on expressive theoretical foundations.

We start with the methods of compiler theory, which is divided into the frontend and backend.

The frontend focuses on the lexing and parsing of source code.

The frontend focuses on the lexing and parsing of source code.

The backend focuses on multilayered translations from an initial IR assembly, to the final target assembly.

Lexing

Lexing

To keep things simple, lexers:

Lexing

To keep things simple, lexers:

- Read source code.

Lexing

To keep things simple, lexers:

- Read source code.
- Translate words into tokens.

Lexing

To keep things simple, lexers:

- Read source code.
- Translate words into tokens.
- Are constructed from regular languages and regular automata.

Parsing

Parsing

To keep things simple, parsers:

Parsing

To keep things simple, parsers:

- Read tokenized source code.

Parsing

To keep things simple, parsers:

- Read tokenized source code.
- Confirm "sentence" structure.

Parsing

To keep things simple, parsers:

- Read tokenized source code.
- Confirm “sentence” structure.
- Translate sentences into IR assembly.

Parsing

To keep things simple, parsers:

- Read tokenized source code.
- Confirm “sentence” structure.
- Translate sentences into IR assembly.
- Are constructed from context free grammars and pushdown automata.

As for backend methods:

As for backend methods:

- Assembly languages are generally implemented using methods derived from register machines.

As for backend methods:

- Assembly languages are generally implemented using methods derived from register machines.
- Such methods coincide well with implementing imperative DSLs.

As for backend methods:

- Assembly languages are generally implemented using methods derived from register machines.
- Such methods coincide well with implementing imperative DSLs.
- Such methods are less effective when implementing functional DSLs.

We need methods that can implement both imperative and functional languages.

We need methods that can implement both imperative and functional languages.

To motivate such methods, let's now take a quick tour of computing history.

Turing machines:

Turing machines:

- Alan Turing, 1936.

Turing machines:

- Alan Turing, 1936.
- Equivalent to μ-recursive functions (math).

Turing machines:

- Alan Turing, 1936.
- Equivalent to μ-recursive functions (math).
- Well suited for modeling theoretical properties of computable functions.

Turing machines:

- Alan Turing, 1936.
- Equivalent to μ-recursive functions (math).
- Well suited for modeling theoretical properties of computable functions.
- Less well suited for modeling practical or performant computable functions.

Lambda calculus $[\lambda x.x]$:

Lambda calculus [$\lambda x.x$]:

- Alonzo Church, 1930s.

Lambda calculus $[\lambda x.x]$:

- Alonzo Church, 1930s.
- Equipotent to Turing machines.

Lambda calculus [$\lambda x.x$]:

- Alonzo Church, 1930s.
- Equipotent to Turing machines.
- Well suited for modeling theoretical grammar of computable functions.

Lambda calculus [$\lambda x.x$]:

- Alonzo Church, 1930s.
- Equipotent to Turing machines.
- Well suited for modeling theoretical grammar of computable functions.
- Less well suited (on its own) for modeling certain *consistency* semantics of computable functions.

LISP programming language:

LISP programming language:

- John McCarthy, late 1950s.

LISP programming language:

- John McCarthy, late 1950s.
- Influenced by the lambda calculus.

LISP programming language:

- John McCarthy, late 1950s.
- Influenced by the lambda calculus.
- Is now a family of languages, including Common Lisp, Scheme, Clojure, and Racket.

LISP programming language:

- John McCarthy, late 1950s.
- Influenced by the lambda calculus.
- Is now a family of languages, including Common Lisp, Scheme, Clojure, and Racket.
- Aligns well with the functional programming paradigm.

To delve further into functional programming, To delve further into functional programming, we need to equip the lambda calculus (or lisp) with a type theory.

To delve further into functional programming, we need to equip the lambda calculus (or lisp) with a type theory.

But first. . .

Set Theory [math]:

*See Russell's Paradox.

Set Theory [math]:

- Georg Cantor, late 1800s.

*See Russell's Paradox.

Set Theory [math]:

- Georg Cantor, late 1800s.
- A foundational language of mathematics.

*See Russell's Paradox.

Set Theory [math]:

- Georg Cantor, late 1800s.
- A foundational language of mathematics.
- Proof that there are "different sizes of infinity."

*See Russell's Paradox.

Set Theory [math]:

- Georg Cantor, late 1800s.
- A foundational language of mathematics.
- Proof that there are “different sizes of infinity.”
- If taken as a naive theory, it leads to contradictions.*

*See Russell’s Paradox.

Type Theory [math]:

Type Theory [math]:

- Bertrand Russell and Alfred North Whitehead, early 1900s.

Type Theory [math]:

- Bertrand Russell and Alfred North Whitehead, early 1900s.
- Principia Mathematica, intended as an alternative to set theory.

Type Theory [math]:

- Bertrand Russell and Alfred North Whitehead, early 1900s.
- Principia Mathematica, intended as an alternative to set theory.
- Mathematicians did not adopt this approach, instead vying for axiomatic set theory.

Type Theory [math]:

- Bertrand Russell and Alfred North Whitehead, early 1900s.
- Principia Mathematica, intended as an alternative to set theory.
- Mathematicians did not adopt this approach, instead vying for axiomatic set theory.
- Helped advance the subject of symbolic logic.

Type Theory [computing]:

† See Lambda Cube.

Type Theory [computing]:

- Multiple contributors (here unnamed), mid 1900s.

†See Lambda Cube.

Type Theory [computing]:

- Multiple contributors (here unnamed), mid 1900s.
- More recently Per Martin-Löf, late 1900s.

† See Lambda Cube.

Type Theory [computing]:

- Multiple contributors (here unnamed), mid 1900s.
- More recently Per Martin-Löf, late 1900s.
- Well suited for modeling certain *consistency* semantics of computable functions.

†See Lambda Cube.

Type Theory [computing]:

- Multiple contributors (here unnamed), mid 1900s.
- More recently Per Martin-Löf, late 1900s.
- Well suited for modeling certain *consistency* semantics of computable functions.
- Aligns well with the lambda calculus†, functional programming, and the family of LISPs.

†See Lambda Cube.

It should be noted:

It should be noted:

Although my current chord and hustle DSLs do not have their own type systems (mostly),

It should be noted:

Although my current chord and hustle DSLs do not have their own type systems (mostly), we don't want to restrict other DSLs from this possibility,

It should be noted:

Although my current chord and hustle DSLs do not have their own type systems (mostly), we don't want to restrict other DSLs from this possibility, and so we need to design for type theory as well.

It should be noted:

Although my current chord and hustle DSLs do not have their own type systems (mostly), we don't want to restrict other DSLs from this possibility, and so we need to design for type theory as well.

The important realization here is that if you equip the lambda calculus (or LISP) with a given type theory, you get a modern functional language such as Haskell.

Type theory or not, this leads us to. . .

Type theory or not, this leads us to…
abstract machines.

Type theory or not, this leads us to...
abstract machines.

We previously introduced this idea, but in the context of the lambda calculus and LISP, we can specifically mention SECD, CESK, and Krivine machines.

Type theory or not, this leads us to...
abstract machines.

We previously introduced this idea, but in the context of the lambda calculus and LISP, we can specifically mention SECD, CESK, and Krivine machines.

Each uses different grammatical artifacts from the untyped lambda calculus to implement its own version of an abstract machine.

Abstract machines

vs

Register machines

Abstract machines:

Abstract machines:

- Consist of some version of a controller, memory lookup, and call stack.

Abstract machines:

- Consist of some version of a controller, memory lookup, and call stack.
- They transition states by updating these components, which is how they perform their computations.

Register machines:

Register machines:

- They are abstract machines.

Register machines:

- They are abstract machines.
- The only difference is their design more closely resembles actual computer architecture.

A final method worth mentioning...

Continuation passing style (CPS):

Continuation passing style (CPS):

A variation on the idea of a function where instead of returning a value,

Continuation passing style (CPS):

A variation on the idea of a function where instead of returning a value, it is passed as the input to another continuation passing function.

Continuation passing style (CPS):

A variation on the idea of a function where instead of returning a value, it is passed as the input to another continuation passing function.

CPS is theoretically sound, and even has its own form of composition:

$$
\begin{aligned}
f(\,x,\ c_1(y)\,) \quad \star \quad & g(\,y,\ c_2(z)\,) \\
:= \quad & f(\,x,\ \lambda y.g(y,\ c_2(z))\,) \\
= \quad & f^\star(\,x,\ c_2(z)\,)
\end{aligned}
$$

(types are hidden for clarity)

Library

In this section I introduce my cctmp library.

‡https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

In this section I introduce my cctmp library.

It is an open source repo on GitHub[‡],

[‡] https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

In this section I introduce my cctmp library.

It is an open source repo on GitHub[‡],
It is not currently ready for official release.

[‡] https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

As for the library itself:

As for the library itself:

- It is implemented using C++17.

As for the library itself:

- It is implemented using C++17.
- It does not use the standard library. (std::)

As for the library itself:

- It is implemented using C++17.
- It does not use the standard library. (std::)
- It is currently split in two: A proof of concept, as well as a semiself hosting version.

As for the proof of concept version:

As for the proof of concept version:

- It is roughly 20 000 lines of code.

As for the proof of concept version:

- It is roughly 20 000 lines of code.
- All DSL code examples I've already shown can be metacompiled in this version.

As for the proof of concept version:

- It is roughly 20 000 lines of code.
- All DSL code examples I've already shown can be metacompiled in this version.
- The hustle DSL does not yet support all language features of the scheme lang (for which it is based).

For example:

```
(define (main n)
  ((if (= n 0) + *) 2 3)
)
```

What about the semiself hosting version?

What about the semiself hosting version?

What is semiself hosting?

What about the semiself hosting version?

What is semiself hosting? One of the long term goals for this metacompiler is to use the chord and hustle DSLs to reimplement parts of the metacompiler itself.

What about the semiself hosting version?

What is semiself hosting? One of the long term goals for this metacompiler is to use the chord and hustle DSLs to reimplement parts of the metacompiler itself.

As this is not proper self hosting (C++ implementing C++), I prefix it with semi- to make the distinction.

As for the semiself hosting version:

As for the semiself hosting version:

- It is a restructuring of the proof of concept version.

As for the semiself hosting version:

- It is a restructuring of the proof of concept version.
- The first 4 000 lines of code represents the bare minimum C++ needed to run meta-assembly programs.

Beyond these two versions, this library is split up into the front end and back end components.

Beyond these two versions, this library is split up into the front end and back end components.

Although the front end comes first in the translation process,

Beyond these two versions, this library is split up into the front end and back end components.

Although the front end comes first in the translation process, it is the backend which shapes the overall design.

Beyond these two versions, this library is split up into the front end and back end components.

Although the front end comes first in the translation process, it is the backend which shapes the overall design. As such, we'll discuss it first.

The backend is designed as an abstract machine,

The backend is designed as an abstract machine, transitioning from state to state:

$$S_0 \to \ldots \to S_n$$

The backend is designed as an abstract machine, transitioning from state to state:

$$S_0 \rightarrow \ldots \rightarrow S_n$$

Each state holds some version of a controller, memory lookup, and call stack.

The backend is designed as an abstract machine, transitioning from state to state:

$$S_0 \to \ldots \to S_n$$

Each state holds some version of a controller, memory lookup, and call stack.

The thing is,

The backend is designed as an abstract machine, transitioning from state to state:

$$S_0 \to \ldots \to S_n$$

Each state holds some version of a controller, memory lookup, and call stack.

The thing is, we don't want to run this abstract machine as an interpreter,

The backend is designed as an abstract machine, transitioning from state to state:

$$S_0 \to \ldots \to S_n$$

Each state holds some version of a controller, memory lookup, and call stack.

The thing is, we don't want to run this abstract machine as an interpreter, the trick in fact is that we use continuation passing functions to represent our machine states.

The twist:

The twist:

Instead of continuation passing, we're continuation constructing.

The twist:

Instead of continuation passing, we're continuation constructing. We're using continuation passing composition to construct our function, which is what we want from a compiler.

The backend then consists of:

The backend then consists of:

- Continuation constructing machines.

The backend then consists of:

- Continuation constructing machines.
- To date, there are 56 distinct machines, making up roughly 1 500 lines of code.

The backend then consists of:

- Continuation constructing machines.
- To date, there are 56 distinct machines, making up roughly 1 500 lines of code.
- Each machine is an indexed *function template*.

By the way,

By the way,
cctmp stands for: continuation constructing template metaprogramming.

By the way,
cctmp stands for: continuation constructing template metaprogramming.

As for how these machines are implemented:

```
template<auto... filler>
struct T_machine<name::go_to, note::id, filler...>
{
    template<machine_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return machine_template(c, ni)
            ::machine_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

```
template<machine_params(c, i, l, t, r), typename... Ts>
constexpr static auto result(Ts... vs)
{






}
```

```cpp
template<auto... filler>
struct T_machine<name::go_to, note::id, filler...>
{
    template<machine_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {






    }
};
```

```
template<auto... filler>
struct T_machine<name::go_to, note::id, filler...>
{




};
```

```
template<machine_params(c, i, l, t, r), typename... Ts>
constexpr static auto result(Ts... vs)
{



}
```

```
template<machine_params(c, i, l, t, r), typename... Ts>
constexpr static auto result(Ts... vs)
{
    constexpr auto ni = dispatch<c>::pos(i);

    return machine_template(c, ni)
        ::machine_result_ts(c, ni, l, t, r, Ts...)
            (vs...);
}
```

```
return machine_template(c, ni)
    ::machine_result_ts(c, ni, l, t, r, Ts...)
        (vs...);
```

```
template<machine_params(c, i, l, t, r), typename... Ts>
constexpr static auto result(Ts... vs)
{


    return machine_template(c, ni)
        ::machine_result_ts(c, ni, l, t, r, Ts...)
            (vs...);
}
```

```
template<machine_params(c, i, l, t, r), typename... Ts>
constexpr static auto result(Ts... vs)
{
    constexpr auto ni = dispatch<c>::pos(i);




}
```

```
template<auto... filler>
struct T_machine<name::go_to, note::id, filler...>
{


        constexpr auto ni = dispatch<c>::pos(i);




};
```

```
template<auto... filler>
struct T_machine<name::go_to, note::id, filler...>
{
    template<machine_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return machine_template(c, ni)
            ::machine_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

What about the controller, memory lookup, and call stack?

What about the controller, memory lookup, and call stack?

- controller: It is passed along as a template parameter.

What about the controller, memory lookup, and call stack?

- controller: It is passed along as a template parameter.
- memory lookup: It is passed along the variadic pack.

What about the controller, memory lookup, and call stack?

- controller: It is passed along as a template parameter.
- memory lookup: It is passed along the variadic pack.
- call stack: It is also passed along the variadic pack.

```
template<machine_params(c, i, l, t, r), typename... Ts>
constexpr static auto result(Ts... vs)
{
    // ...

    return machine_template(c, i)
        ::machine_result_ts(c, i, l, t, r, Ts...)
            (vs...);
}
```

```
template<machine_params(c, i, l, t, r), typename... Ts>
constexpr static auto result(Ts... vs)
{
    // ...

    return machine_template(c, i)
        ::machine_result_ts(c, i, l, t, r, Ts...)
            (vs...);
}
```

```
template<machine_params(c, i, l, t, r), typename... Ts>
constexpr static auto result(Ts... vs)
{
    // ...

    return machine_template(c, i)
        ::machine_result_ts(c, i, l, t, r, Ts...)
            (vs...);
}
```

As for the controller:

As for the controller:

- It is a program composed of meta-assembly.

As for the controller:

- It is a program composed of meta-assembly.
- It is implemented as an array of instructions.

As for the controller:

- It is a program composed of meta-assembly.
- It is implemented as an array of instructions.
- Each instruction is an array of unsigned integers.

What does this assembly look like?

What does this assembly look like?

We saw it once before:

```
constexpr size_type value[][8] =
{
    { MN::id          , MT::id        ,  0,   0,   0,   0,   0,   1 },
    { MN::hash        , MT::port      ,  5,   0,   0,   0,   0,   1 },
    { MN::pad         , MT::select    ,  0,   1,   0,   0,   0,   1 },
    { MN::pad         , MT::id        ,  0,   0,   0,   0,   0,   1 },
    { MN::go_to       , MT::id        , 50,   0,   0,   0,   0,   1 },
    { MN::id          , MT::id        ,  0,   0,   0,   0,   0,   1 },
    { MN::eval        , MT::back      ,  7,   0,   0,   0,   0,   4 },
    { MN::id          , MT::id        ,  0,   0,   0,   0,   0,   1 },
    { MN::lookup      , MT::first     ,  0,   7,   0,   0,   0,   1 },
    { MN::halt        , MT::first     ,  0,   0,   0,   0,   0,   1 },
    { MN::eval        , MT::back      , 11,   0,   0,   0,   0,   5 },
    { MN::id          , MT::id        ,  0,   0,   0,   0,   0,   1 },
    { MN::arg         , MT::select    ,  1,   0,   0,   0,   0,   1 },
    { MN::arg         , MT::drop      ,  0,   0,   0,   0,   0,   1 },
    { MN::halt        , MT::first     ,  0,   0,   0,   0,   0,   1 },
    { MN::type        , MT::n_number  ,  0,   0,   0,   0,   0,   1 },
    { MN::literal     , MT::back      ,  0,   0,   0,   0,   0,   1 },
```

As for the memory lookup and call stack:

As for the memory lookup and call stack:

Within the context of a variadic pack, and for clarity,

As for the memory lookup and call stack:

Within the context of a variadic pack, and for clarity, I use the term universe for the memory lookup,

As for the memory lookup and call stack:

Within the context of a variadic pack, and for clarity,
I use the term universe for the memory lookup,
and the term stage for the call stack.

As for the universe:

As for the universe:

- It is implemented using the left side of the variadic pack.

As for the universe:

- It is implemented using the left side of the variadic pack.
- New values are inserted at the end of the left side of the pack.

As for the stage:

As for the stage:

- It is implemented using the right side of the variadic pack.

As for the stage:

- It is implemented using the right side of the variadic pack.
- New values are pushed to the back of the pack.

As for the stage:

- It is implemented using the right side of the variadic pack.
- New values are pushed to the back of the pack.
- Argument order is preserved when applying functions to their values.

This concludes our discussion of the back end.

This concludes our discussion of the back end.

We continue now with the front end.

metacompiler frontend:

§https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools

metacompiler frontend:

- The component design and implementation is inspired by the "Dragon Book".[§]

[§] https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools

metacompiler frontend:

- The component design and implementation is inspired by the "Dragon Book".[§]
- A constexpr LL(1) parser generator is used to construct transition tables for DSL context free grammars.

[§]https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools

metacompiler frontend:

- The component design and implementation is inspired by the "Dragon Book".[§]
- A constexpr LL(1) parser generator is used to construct transition tables for DSL context free grammars.
- The parser generator is designed analogous to the DSLs, where you take a string literal as input.

[§]https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

            "Start",

        // hustle:

            "Start   -> ( Generic )                  ;"
            "Generic -> type Param Params ( Main ) ;"
            "        -> Main                       ;"
            "Params  -> Param Params               ;"
            "        -> empty                      ;"
            "Param   -> identifier : param_type    ;"

            // main:
```

As for a lexer generator?

As for a lexer generator?

Not yet supported.

As for a lexer generator?

Not yet supported.
On my list of things to do.

As for a lexer generator?

Not yet supported.
On my list of things to do.

Currently each DSL handcodes its own lexer, based on automata theory.

Next, we need to discuss how we map DSL source code variables

¶Structure and Interpretation of Computer Programs.

Next, we need to discuss how we map DSL source code variables onto continuation constructing universes

¶Structure and Interpretation of Computer Programs.

Next, we need to discuss how we map DSL source code variables onto continuation constructing universes (the variadic pack of our state machines).

¶Structure and Interpretation of Computer Programs.

Next, we need to discuss how we map DSL source code variables onto continuation constructing universes (the variadic pack of our state machines).

I borrow a design from the classic SICP[¶] textbook,

[¶] Structure and Interpretation of Computer Programs.

Next, we need to discuss how we map DSL source code variables onto continuation constructing universes (the variadic pack of our state machines).

I borrow a design from the classic SICP[¶] textbook, which teaches how to build a metacircular evaluator.

[¶] Structure and Interpretation of Computer Programs.

Next, we need to discuss how we map DSL source code variables onto continuation constructing universes (the variadic pack of our state machines).

I borrow a design from the classic SICP[¶] textbook, which teaches how to build a metacircular evaluator. It uses what's called an environment to keep track of its variables.

¶Structure and Interpretation of Computer Programs.

An environment is an associative array, where:

An environment is an associative array, where:

- Each environment is a list of frames.

An environment is an associative array, where:

- Each environment is a list of frames.
- Each frame is a list of bindings.

An environment is an associative array, where:

- Each environment is a list of frames.
- Each frame is a list of bindings.
- Each binding is a variable/value pair.

In our case when we parse the source code of a DSL, In our case when we parse the source code of a DSL, every time we encounter a variable we assign it an index within the universe.

In our case when we parse the source code of a DSL, every time we encounter a variable we assign it an index within the universe. This works as long as we also have a stack to keep record of locally defined universe/stage indices during function calls.

In our case when we parse the source code of a DSL, every time we encounter a variable we assign it an index within the universe. This works as long as we also have a stack to keep record of locally defined universe/stage indices during function calls.

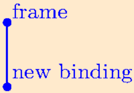
With that in mind, we have the following illustration:

environment

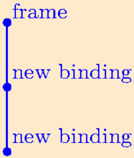
environment

- frame

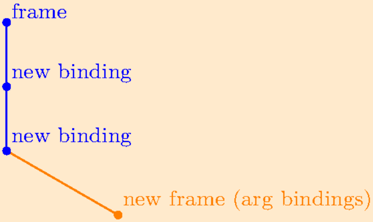
environment



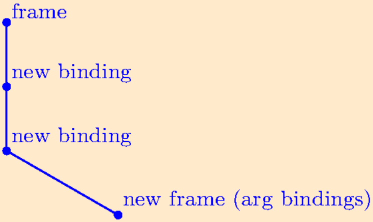
environment



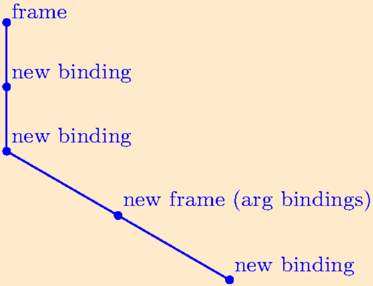
environment



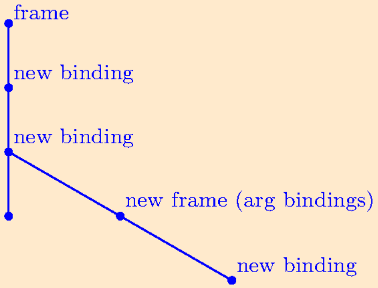
environment



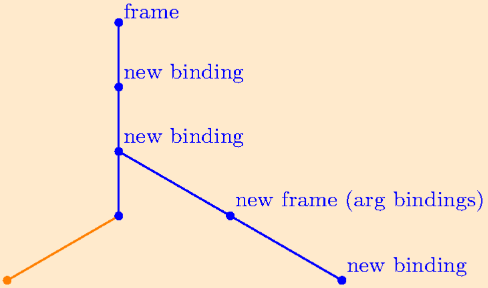
environment



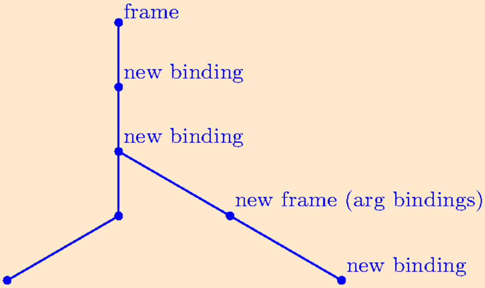
environment



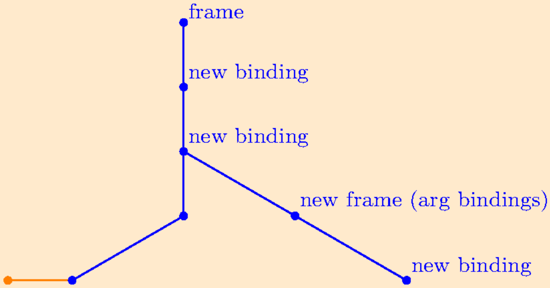
environment



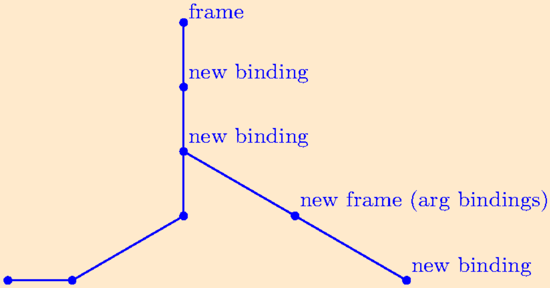
environment



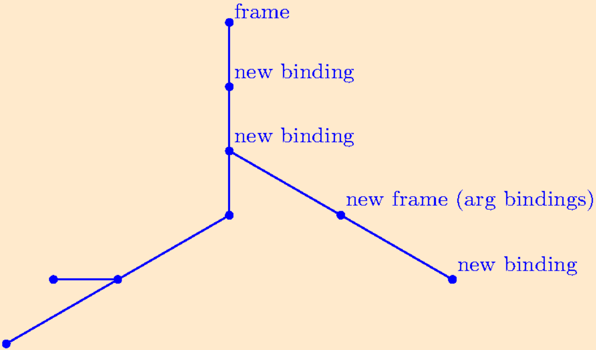
environment



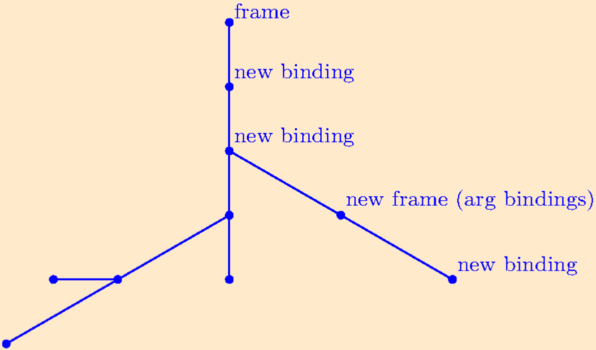
environment



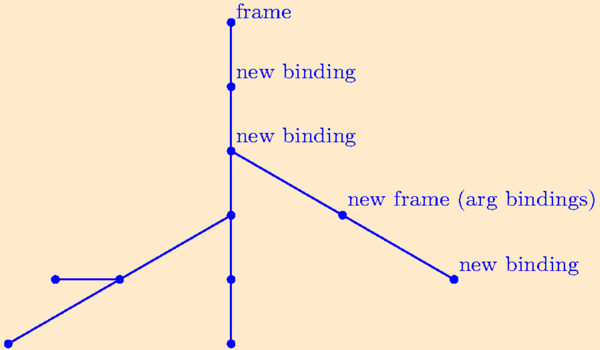
environment



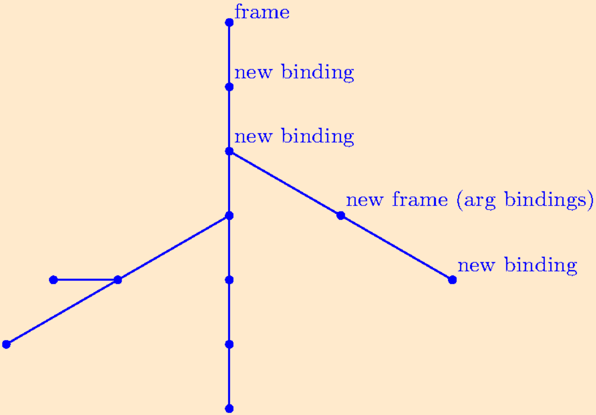
environment



environment



environment



To summarize the parsing of variables:

To summarize the parsing of variables:

- The metacompiler frontend uses a SICP-style environment to keep track of variables.

To summarize the parsing of variables:

- The metacompiler frontend uses a SICP-style environment to keep track of variables.
- Instead of keeping track of values, it holds an index (a promise) of where those values will eventually be in the continuation constructing universe.

To summarize the parsing of variables:

- The metacompiler frontend uses a SICP-style environment to keep track of variables.
- Instead of keeping track of values, it holds an index (a promise) of where those values will eventually be in the continuation constructing universe.
- This is why the meta-assembly controller consists of numerical content only.

Before we finish this section,

Before we finish this section,

We have two final technical issues to address,

Before we finish this section,

We have two final technical issues to address,

Which are of fundamental importance:

Before we finish this section,

We have two final technical issues to address,

Which are of fundamental importance:

- Recursion.

Before we finish this section,
We have two final technical issues to address,
Which are of fundamental importance:

- Recursion.
- Well definedness.

As for recursion?

Because of how we index variables,
we actually know the history of a given universe:

Because of how we index variables,
we actually know the history of a given universe:

Anything further right in the universe was defined later in the program.

Because of how we index variables,
we actually know the history of a given universe:

Anything further right in the universe was defined later in the program. Mutability can complicate this, but otherwise when we define a function—even a recursive one—we only need the left side of the universe at the time the function was defined.

Because of how we index variables,
we actually know the history of a given universe:

Anything further right in the universe was defined later in the program. Mutability can complicate this, but otherwise when we define a function—even a recursive one—we only need the left side of the universe at the time the function was defined.

When we apply a function call,

Because of how we index variables,
we actually know the history of a given universe:

Anything further right in the universe was defined later in the program. Mutability can complicate this, but otherwise when we define a function—even a recursive one—we only need the left side of the universe at the time the function was defined.

When we apply a function call, we must provide that leftside universe along with the function arguments.

Even though our functions are continuation passing,

Even though our functions are continuation passing, they are still functions:

Even though our functions are continuation passing, they are still functions:

If we call the same function template with the same type input,

Even though our functions are continuation passing, they are still functions:

If we call the same function template with the same type input, the C++ compiler will recognize it as a recursive call and so complete the function.

Even though our functions are continuation passing, they are still functions:

If we call the same function template with the same type input, the C++ compiler will recognize it as a recursive call and so complete the function.

This is why recursion works.

As for well-definedness?

As for well-definedness?

We need to be sure that each meta-assembly controller which compiles,

As for well-definedness?

We need to be sure that each meta-assembly controller which compiles, also corresponds to *exactly one* abstract machine.

As for well-definedness?

We need to be sure that each meta-assembly controller which compiles, also corresponds to *exactly one* abstract machine. This is a question of ambiguity.

As for well-definedness?

We need to be sure that each meta-assembly controller which compiles, also corresponds to *exactly one* abstract machine. This is a question of ambiguity.

When we move to the next continuation, does it correspond with the next instruction?

The only time this might *not* happen is when the C++ compiler recognizes a recursive call, The only time this might *not* happen is when the C++ compiler recognizes a recursive call, that the coder didn't intend.

The only time this might *not* happen is when the C++ compiler recognizes a recursive call, that the coder didn't intend.

In that case:

$$\begin{tikzcd}
S_0 \arrow[r] & S_1 \arrow[r] & \dots \arrow[r] & S_{N-1} \arrow[r] \arrow[lll, bend right] & S_N \arrow[r] & S_{N+1}
\end{tikzcd}$$

We're at step $N - 1$ and we think we're moving to step N, when in fact the compiler creates a recursive call and moves us to step 0.

Because of this ($S_N == S_0$),
but we have no way of knowing if ($S_{N+1} == S_1$).

If we had intended a recursive call, then these paths line up and there's no problem.

If we had intended a recursive call, then these paths line up and there's no problem.

If not, no worries, this case doesn't show up in practice:

If we had intended a recursive call, then these paths line up and there's no problem.

If not, no worries, this case doesn't show up in practice: As our metacompiler continuation constructs, it moves forward along the controller, and so calls the next function template with an entirely new controller index.

If we had intended a recursive call, then these paths line up and there's no problem.

If not, no worries, this case doesn't show up in practice: As our metacompiler continuation constructs, it moves forward along the controller, and so calls the next function template with an entirely new controller index. Two C++ functions are considered different if they have different template parameters.

Is it possible to move backward in the controller?

Is it possible to move backward in the controller?

Yes, but that requires an explicit jump instruction.

Is it possible to move backward in the controller?

Yes, but that requires an explicit jump instruction. The chord and hustle DSLs require the user explicitly request a jump as a goto, a branch, or a recursive call.

Finally, there is still the possibility that the user asks for a goto or branch,

Finally, there is still the possibility that the user asks for a goto or branch, but the C++ compiler recognizes it as a recursive call.

Finally, there is still the possibility that the user asks for a goto or branch, but the C++ compiler recognizes it as a recursive call.

In this case the compiler also requires the return type of the function being defined since our continuation machines use auto deduction.

Finally, there is still the possibility that the user asks for a goto or branch, but the C++ compiler recognizes it as a recursive call.

In this case the compiler also requires the return type of the function being defined since our continuation machines use auto deduction. As we did not explicitly provide that information, the C++ compiler will then halt with its own error message.

Entailment

This metacompiler paradigm has now been properly introduced.

This metacompiler paradigm has now been properly introduced.

There are several consequences worth discussing,

This metacompiler paradigm has now been properly introduced.

There are several consequences worth discussing, but as this talk winds down I will instead leave you with what I hope is a thought provoking idea.

It is safe to say that C++ has problems as of late:

It is safe to say that C++ has problems as of late: ABI.

It is safe to say that C++ has problems as of late: ABI. Safety.

It is safe to say that C++ has problems as of late: ABI. Safety. Successor Languages.

It is safe to say that C++ has problems as of late: ABI. Safety. Successor Languages.

C++ has also become so successful that it now has an oversaturated grammar space.

It is safe to say that C++ has problems as of late: ABI. Safety. Successor Languages.

C++ has also become so successful that it now has an oversaturated grammar space. This metacompiler paradigm, with its DSL engine, can help mitigate these problems.

I come from a pure math background and have spent a lot of time on foundational languages.

I come from a pure math background and have spent a lot of time on foundational languages.

Foundations are all about design,

I come from a pure math background and have spent a lot of time on foundational languages.

Foundations are all about design, and after spending time on programming language design,

I come from a pure math background and have spent a lot of time on foundational languages.

Foundations are all about design, and after spending time on programming language design, I'm willing to say math has things to learn from software engineering, but that software engineering also has a few things it can learn from pure math.

Math,

‖ See Algebraic Topology, Algebraic Geometry, Analytic Combinatorics, etc.

Math, like computing,

‖ See Algebraic Topology, Algebraic Geometry, Analytic Combinatorics, etc.

Math, like computing, has many domain specific languages.

‖See Algebraic Topology, Algebraic Geometry, Analytic Combinatorics, etc.

Math, like computing, has many domain specific languages. It's not so concerned with performance, but it doesn't have interoperability concerns between languages either.[‖]

[‖] See Algebraic Topology, Algebraic Geometry, Analytic Combinatorics, etc.

Math, like computing, has many domain specific languages. It's not so concerned with performance, but it doesn't have interoperability concerns between languages either.‖

The reason for this is that math has already found ways to mitigate its consistency problems.

‖See Algebraic Topology, Algebraic Geometry, Analytic Combinatorics, etc.

Math, like computing, has many domain specific languages. It's not so concerned with performance, but it doesn't have interoperability concerns between languages either.‖

The reason for this is that math has already found ways to mitigate its consistency problems. It chooses to seek out foundational languages (such as Set Theory) to implement all other languages.

‖See Algebraic Topology, Algebraic Geometry, Analytic Combinatorics, etc.

The fact that C++ can effectively model all other programming languages means it is a foundational language of computing.

The fact that C++ can effectively model all other programming languages means it is a foundational language of computing.

Any C++ successor language should directly support this metaprogramming paradigm.

Why?

Why?

If it's truly a successor lang, it will eventually have the same grammar oversaturation problem.

Why?

If it's truly a successor lang, it will eventually have the same grammar oversaturation problem.

Oversaturation comes from many communities using the language, each with their own overlapping but distinct domains of interest.

A metacompiler successor lang can mitigate over-saturation with a DSL engine.

A metacompiler successor lang can mitigate over-saturation with a DSL engine.

Such a lang can certainly still mitigate safety concerns with type checkers and borrow checkers,

A metacompiler successor lang can mitigate over-saturation with a DSL engine.

Such a lang can certainly still mitigate safety concerns with type checkers and borrow checkers, but it can also mitigate safety with well designed and standardized DSLs, A metacompiler successor lang can mitigate oversaturation with a DSL engine.

Such a lang can certainly still mitigate safety concerns with type checkers and borrow checkers, but it can also mitigate safety with well designed and standardized DSLs, especially if it already provides that support.

End

(thank you)

Questions?

example:
array square (map)

```
main out in end                            ;

vars:                                      ;
  declare arr_sq                           ;
defs:                                      ;
  arr_sq # map[1]{square||} [) [,)         ;
body:                                      ;
  . = arr_sq !out in end                   ;
  return _                                 ;

, binding( "square" , _square_ )
```

example:
dot product (fold)

```
main out in end in1

vars:
  declare dot_prod
defs:
  dot_prod # fold[2]{add * @|multiply||} <> [,)
body:
  . = dot_prod !out in end in1
  return _
```

example:
convolution (fold)

```
main out in end in1

vars:
  declare conv
defs:
  conv # fold[2]{add * @|multiply||} <> (-|+,]
body:
  . = conv !out end in1
  return _
```

example:
void effects (mutability):

```
main x y                ;

body:                   ;
  . = increment y       ;
  void appoint !x _     ;
  return x              ;
```

example:

semidynamic typing (deferred type checking):

```
type T                  ;
main c n                ;

body:                   ;
  test equal c _1_0i    ;
  branch set_c_to_five  ;
  c = increment n       ;
  return c              ;

set_c_to_five:          ;
  c # 5:T               ;
  return c              ;
```

```
, binding("_1_0i", complex_number{1, 0})
```