A universal data structure for compile time use

Daniel Nikpayuk

2025

Copyright 2025 Daniel Nikpayuk

Outline

Outline

Six parts:

Outline

Six parts:

1. Introduction
2. Motivation
3. Interrogation

Outline

Six parts:

1. Introduction
2. Motivation
3. Interrogation
4. Method Equip
5. Proof Assistant
6. Applications

Introduction

Who am I?

Who am I?

- I'm a self-taught coder.
- I've been programming in C++ since 2005.

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.

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++?

Why C++?

- It is a life goal of mine to build a programming language for multimedia production.

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

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

This talk is inspired by the talk I gave last year (2024) at this conference, This talk is inspired by the talk I gave last year (2024) at this conference, titled:

C++ is a Metacompiler

The main idea behind that talk

The main idea behind that talk was to demonstrate a theory based metaprogramming technique

The main idea behind that talk was to demonstrate a theory based metaprogramming technique which in practice allows us to inject function code

The main idea behind that talk was to demonstrate a theory based metaprogramming technique which in practice allows us to inject function code into the compiler's syntax tree.

In effect,

In effect,

C++ compilers can create constexpr functions

In effect,

C++ compilers can create constexpr functions from string literals

In effect,

C++ compilers can create constexpr functions from string literals to be used either at compile time or at run time.

In effect,

C++ compilers can create constexpr functions from string literals to be used either at compile time or at run time.

I call a compiler with such an ability a meta-compiler.

Although this talk is not intended to be a direct continuation of last year's

Although this talk is not intended to be a direct continuation of last year's, it is still related as its techniques are used in the designs presented here.

Although this talk is not intended to be a direct continuation of last year's, it is still related as its techniques are used in the designs presented here.

As such, I offer a quick summary.

Although this talk is not intended to be a direct continuation of last year's, it is still related as its techniques are used in the designs presented here.

As such, I offer a quick summary. In particular, a metacompiler is such that it turns string literals like this:

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

or 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 meta-assembly
that looks like 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 give to continuation passing machines that look like this:

```
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 constexpr functions.

In any case,

In any case,

The underlying motivation for last year's talk, as well as my onging project (library)

In any case,

The underlying motivation for last year's talk, as well as my onging project (library) is to create compile time tools

In any case,

The underlying motivation for last year's talk, as well as my onging project (library) is to create compile time tools to build and translate arbitrary DSLs into constexpr functions.

As for this year's talk:

As for this year's talk:

It's about writing code beyond the limitations of constexpr 17–23.

As for this year's talk:

It's about writing code beyond the limitations of constexpr 17–23. Two limitations notably being a lack of heap allocation,

As for this year's talk:

It's about writing code beyond the limitations of constexpr 17–23. Two limitations notably being a lack of heap allocation, as well as the general lack of available data structures.

As for this year's talk:

It's about writing code beyond the limitations of constexpr 17–23. Two limitations notably being a lack of heap allocation, as well as the general lack of available data structures.

This talk is about the ability to create arbitrary compile time data structures

As for this year's talk:

It's about writing code beyond the limitations of constexpr 17–23. Two limitations notably being a lack of heap allocation, as well as the general lack of available data structures.

This talk is about the ability to create arbitrary compile time data structures under these constraints.

Interrogation

Why interrogate?

Why interrogate?

When I hear the word “interrogation”

Why interrogate?

When I hear the word “interrogation” I think of tv shows with detectives

Why interrogate?

When I hear the word "interrogation" I think of tv shows with detectives and rooms where they question people.

Here we interrogate the purpose of this talk.

Here we interrogate the purpose of this talk.

In this conference room.

Here we interrogate the purpose of this talk.

In this conference room.
We're all detectives now.

The first question to ask would be:

The first question to ask would be:

Why A universal data structure for compile time use?

The first question to ask would be:

Why A universal data structure for compile time use?

Why this title?

The next questions to ask are:

The next questions to ask are:

- What makes a data structure universal?

The next questions to ask are:

- What makes a data structure universal?
- What is compile time, really?

The next questions to ask are:

- What makes a data structure universal?
- What is compile time, really?
- Why make a universal data structure

The next questions to ask are:

- What makes a data structure universal?
- What is compile time, really?
- Why make a universal data structure specifically for compile time use?

First, we ask:

First, we ask:

What makes a data structure universal?

The simple answer:

The simple answer:

A data structure is universal if it can simulate all possible data structures.

In this case,

In this case,

And although there might still be alternative approaches one can take,

In this case,

And although there might still be alternative approaches one can take, the solution I am presenting here is that of a type system.

i.e.

i.e.

A mathematical type system

i.e.

A mathematical type system with

i.e.

A mathematical type system with pairs,

i.e.

A mathematical type system with pairs, disjoint unions,

i.e.

A mathematical type system with pairs, disjoint unions, lists,

i.e.

A mathematical type system with pairs, disjoint unions, lists, among other types.

i.e.

A mathematical type system with pairs, disjoint unions, lists, among other types.

I choose this design because such type systems are well understood,

i.e.

A mathematical type system with pairs, disjoint unions, lists, among other types.

I choose this design because such type systems are well understood, theoretically sound,

i.e.

A mathematical type system with pairs, disjoint unions, lists, among other types.

I choose this design because such type systems are well understood, theoretically sound, and have been known to successfully simulate all known varieties of structured data.

Following this, we ask:

Following this, we ask:

What is compile time,

Following this, we ask:

What is compile time, really?

We ask this question

We ask this question because the designs in this talk require

We ask this question because the designs in this talk require a more refined understanding of compile time

We ask this question because the designs in this talk require a more refined understanding of compile time than what many C++ programmers might be use to.

First,

First, we require a refined notion of time.

We ask:

We ask:

- What is a timescape?

We ask:

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

The short answer:

The short answer:

When observing the lifespan of a program,

The short answer:

When observing the lifespan of a program, a timescape

The short answer:

When observing the lifespan of a program, a timescape allows us to decompose said lifespan

The short answer:

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

As for specific timescopes,

As for specific timescopes,
in words they are summarized as:

As for specific timescopes,
in words they are summarized as:

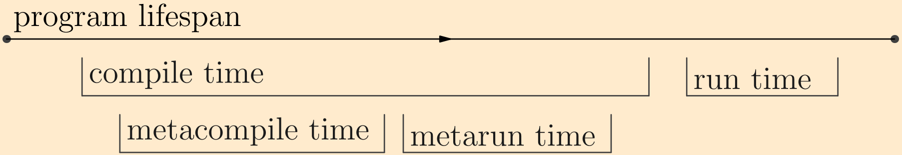
- Run time
- Compile time
- Metarun time
- Metacompile time

As for specific timescopes,
in words they are summarized as:

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

Copyright 2025 Daniel Nikpayuk

In graphical terms they are summarized as:


program lifespan
compile time
run time
metacompile time
metarun time

Returning back to our initial interrogation,

Returning back to our initial interrogation,
we finally ask:

Returning back to our initial interrogation,
we finally ask:

Why make a universal data structure

Returning back to our initial interrogation,
we finally ask:

Why make a universal data structure
specifically for compile time use?

The simple answer:

The simple answer:

Non-transient constexpr allocation.

The not so simple answer:

The not so simple answer:

Non-transient constexpr allocation.

The not so simple answer:

Non-transient constexpr allocation. (p2670r0)

I have a story. . .

I have a story. . .

I was working on my compile time project, I have a story...

I was working on my compile time project, and I kept running into the same problem.

I have a story. . .

I was working on my compile time project, and I kept running into the same problem. The compiler (GCC, Clang) kept telling me I couldn't return constexpr data structures

I have a story. . .

I was working on my compile time project, and I kept running into the same problem. The compiler (GCC, Clang) kept telling me I couldn't return constexpr data structures from a function if those data structures

I have a story...

I was working on my compile time project, and I kept running into the same problem. The compiler (GCC, Clang) kept telling me I couldn't return constexpr data structures from a function if those data structures had member values that were pointers.

I have a story. . .

I was working on my compile time project, and I kept running into the same problem. The compiler (GCC, Clang) kept telling me I couldn't return constexpr data structures from a function if those data structures had member values that were pointers.

For example:

```
template<typename T, auto N>
class inplace_vector
{
   private:

     T initial[N];
     T* terminal;

   public:

     constexpr inplace_vector() :
       initial{}, terminal{initial} { }
};
```

```
template<typename T, auto N>
class inplace_vector
{
   private:

     T initial[N];
     T* terminal;

   public:

     constexpr inplace_vector() :
       initial{}, terminal{initial} { }
};
```

```
template<typename T, auto N>
class inplace_vector
{
   private:

     T initial[N];
     T* terminal;

   public:

     constexpr inplace_vector() :
       initial{}, terminal{initial} { }
};
```

```
template<typename T, auto N>
class inplace_vector
{
   private:

     T initial[N];
     T* terminal;

   public:

     constexpr inplace_vector() :
       initial{}, terminal{initial} { }
};
```

```
template<typename T, auto N>
constexpr auto make_inplace_vector()
  { return inplace_vector<T, N>{}; }


int main()
{
   constexpr auto vector_0 =
     make_inplace_vector<int, 5>();
       // error: pointer to subobject is
       //        not a constant expression

   return 0;
}
```

```
template<typename T, auto N>
constexpr auto make_inplace_vector()
  { return inplace_vector<T, N>{}; }


int main()
{
   constexpr auto vector_0 =
     make_inplace_vector<int, 5>();
       // error: pointer to subobject is
       //        not a constant expression

   return 0;
}
```

```
template<typename T, auto N>
constexpr auto make_inplace_vector()
  { return inplace_vector<T, N>{}; }


int main()
{
   constexpr auto vector_0 =
     make_inplace_vector<int, 5>();
       // error: pointer to subobject is
       //        not a constant expression

   return 0;
}
```

```
template<typename T, auto N>
constexpr auto make_inplace_vector()
  { return inplace_vector<T, N>{}; }


int main()
{
   constexpr auto vector_0 =
     make_inplace_vector<int, 5>();
       // error: pointer to subobject is
       //        not a constant expression

   return 0;
}
```

```
template<typename T, auto N>
constexpr auto make_inplace_vector()
  { return inplace_vector<T, N>{}; }


int main()
{
   constexpr auto vector_0 =
     make_inplace_vector<int, 5>();
       // error: pointer to subobject is
       //         not a constant expression

   return 0;
}
```

After much trial and error,

After much trial and error, I realized I had no solution.

After much trial and error, I realized I had no solution.

I eventually came to the conclusion

After much trial and error, I realized I had no solution.

I eventually came to the conclusion the best way forward was to work around the problem itself

After much trial and error, I realized I had no solution.

I eventually came to the conclusion the best way forward was to work around the problem itself by designing a library of compile time containers that didn't have member pointers.

I asked around, and confirmed it.

I asked around, and confirmed it.

It is a known problem.

I asked around, and confirmed it.

It is a known problem. In particular Hana Dusíková told me to look up the term non-transient constexpr allocation (thanks Hana!).

Barry Revzin's blog

Barry Revzin's blog titled
What's so hard about constexpr allocation?
explains it well.

Barry Revzin's blog titled
What's so hard about constexpr allocation?
explains it well.

Without going into detail,

Barry Revzin's blog titled
What's so hard about constexpr allocation?
explains it well.

Without going into detail, if we allowed general purpose constexpr allocation,

Barry Revzin's blog titled
What's so hard about constexpr allocation?
explains it well.

Without going into detail, if we allowed general purpose constexpr allocation, there would be code compilations where for example the compiler would allocate an object at compile time,

Barry Revzin's blog titled
What's so hard about constexpr allocation?
explains it well.

Without going into detail, if we allowed general purpose constexpr allocation, there would be code compilations where for example the compiler would allocate an object at compile time, but then deallocate it at run time leading to serious bugs.

To prevent this,

To prevent this, constexpr allocation is currently restricted

To prevent this, constexpr allocation is currently restricted disallowing pointers as mentioned earlier.

To answer our interrogative question then:

To answer our interrogative question then:

We are making a universal data structure specifically for compile time use

To answer our interrogative question then:

We are making a universal data structure specifically for compile time use in large part to work around the constexpr allocation problem,

To answer our interrogative question then:

We are making a universal data structure specifically for compile time use in large part to work around the constexpr allocation problem, and to provide a library of general purpose containers

To answer our interrogative question then:

We are making a universal data structure specifically for compile time use in large part to work around the constexpr allocation problem, and to provide a library of general purpose containers to help us solve more complex problems we may encounter at compile time.

Method Equip

What is method equip?

The intuition is this:

The intuition is this:

You start with a given data structure

The intuition is this:

You start with a given data structure and temporarily equip it

The intuition is this:

You start with a given data structure and temporarily equip it with additional class methods.

The intuition is this:

You start with a given data structure and temporarily equip it with additional class methods.

You can then talk about data objects of that class

The intuition is this:

You start with a given data structure and temporarily equip it with additional class methods.

You can then talk about data objects of that class using relevent perspectives restricted to *local scopes* of interest.

For example,

For example, what if we started with a vector, For example, what if we started with a vector, and wanted to temporarily equip it with specialized versions of its push method?

For example, what if we started with a vector, and wanted to temporarily equip it with specialized versions of its push method?

```
some_vector.push(value);
```

There are situations where we want to use the builtin push, There are situations where we want to use the builtin push, but other times we might want to push to a vector

There are situations where we want to use the builtin push, but other times we might want to push to a vector only if the value or object is *unique*.

If we also want to keep the original name, we would call this:

If we also want to keep the original name, we would call this:

Contextual thinking,

If we also want to keep the original name, we would call this:

Contextual thinking, or context switching.

That’s the intuition,

That's the intuition, but for our purposes we choose method equip

That's the intuition, but for our purposes we choose method equip as a way to solve our constexpr allocation problem.

The very idea of modularizing out most of a container's content

The very idea of modularizing out most of a container's content to local scopes

The very idea of modularizing out most of a container's content to local scopes means we can also factor out

The very idea of modularizing out most of a container's content to local scopes means we can also factor out the use of member pointers

The very idea of modularizing out most of a container's content to local scopes means we can also factor out the use of member pointers to those contexts.

The very idea of modularizing out most of a container's content to local scopes means we can also factor out the use of member pointers to those contexts.

This is our way around constexpr allocations.

So how do we achieve this sort of paradigm in C++?

To start,

To start, by reimplementing containers such as array and vector.

To start, by reimplementing containers such as array and vector. We do so because the standard versions do not naturally support method equip.

To start, by reimplementing containers such as array and vector. We do so because the standard versions do not naturally support method equip.

In particular,

To start, by reimplementing containers such as array and vector. We do so because the standard versions do not naturally support method equip.

In particular, we give the container (re)definitions a specific method:

```
template<typename Lens>
constexpr auto equip() -> Lens
  { return facade_type{this}; }
```

```
template<typename Lens>
constexpr auto equip() -> Lens
  { return facade_type{this}; }
```

```
template<typename Lens>
constexpr auto equip() -> Lens
  { return facade_type{this}; }
```

```
template<typename Lens>
constexpr auto equip() -> Lens
  { return facade_type{this}; }
```

```
template<typename Lens>
constexpr auto equip() -> Lens
  { return facade_type{this}; }
```

This is to say,

This is to say,

We use a template parameter so as to defer our construction,

This is to say,

We use a template parameter so as to defer our construction, but when we do construct,

This is to say,

We use a template parameter so as to defer our construction, but when we do construct, we inherit.

I use the word lens

I use the word lens because we are *equipping* our memory model with methods, I use the word lens because we are *equipping* our memory model with methods, but those methods are themselves bound within a lens class.

This paradigm actually has two specific versions of *equip*, This paradigm actually has two specific versions of *equip*, the second one being needed to handle *constness*:

```
template<typename CLens>
constexpr auto cequip() -> CLens
{
   return
     cfacade_type{static_cast<model const*>(this)};
}
```

```
template<typename CLens>
constexpr auto cequip() -> CLens
{
   return
     cfacade_type{static_cast<model const*>(this)};
}
```

As for lens classes:

As for lens classes:

They hold related methods together

As for lens classes:

They hold related methods together, but they also have to refer to data within our memory models of interest.

As for lens classes:

They hold related methods together, but they also have to refer to data within our memory models of interest.

For performance reasons this suggests shallow copies, or pointers.

This is how we avoid constexpr allocation in practice,

This is how we avoid constexpr allocation in practice, by instantiating these lenses within local scopes only.

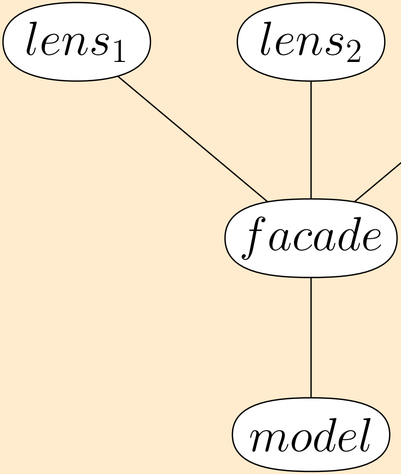
This is how we avoid constexpr allocation in practice, by instantiating these lenses within local scopes only.

As objects they'll be destroyed when their lifetimes end,

This is how we avoid constexpr allocation in practice, by instantiating these lenses within local scopes only.

As objects they'll be destroyed when their lifetimes end, and there will be no need to return them directly from functions.

To summarize method equip as a graphic:



-inherits facade

-pointer to model

-no pointers
-compile time compatible
-method equip

We haven't discussed facades yet.

We haven't discussed facades yet.

In that case,

We haven't discussed facades yet.

In that case, let's go through the relationships of this paradigm in more detail,

We haven't discussed facades yet.

In that case, let's go through the relationships of this paradigm in more detail, starting with the model.

The C memory model is not only a very big array, it is a contiguous array.

If we take C style arrays as our foundation, and if we want to maintain a certain consistency with C++ code, I'm willing to say there are 3 relevant memory models:

If we take C style arrays as our foundation, and if we want to maintain a certain consistency with C++ code, I'm willing to say there are 3 relevant memory models:

1. string literals

If we take C style arrays as our foundation, and if we want to maintain a certain consistency with C++ code, I'm willing to say there are 3 relevant memory models:

1. string literals
2. arrays

If we take C style arrays as our foundation, and if we want to maintain a certain consistency with C++ code, I'm willing to say there are 3 relevant memory models:

1. string literals
2. arrays
3. vectors

To show one such model in code, we have:

```
template<typename Type, typename SizeType, SizeType Size>
class vector_model
{
   public:
         // type aliases go here.
   protected:

     type initial[Size]; // compile time compatible.
     size_type terminal; // compile time compatible.

   public:
     constexpr vector_model() : initial{}, terminal{} { }

     // initial:

        constexpr ctype_ptr cbegin() const { return initial; }
        constexpr  type_ptr  begin()       { return initial; }

     // terminal:

        constexpr size_type size() const { return terminal; }
        constexpr void set_size(size_ctype n) { terminal = n; }
};
```

```
template<typename Type, typename SizeType, SizeType Size>
class vector_model
{
   public:
          // type aliases go here.
   protected:

     type initial[Size]; // compile time compatible.
     size_type terminal; // compile time compatible.

   public:
     constexpr vector_model() : initial{}, terminal{} { }

     // initial:

        constexpr ctype_ptr cbegin() const { return initial; }
        constexpr  type_ptr  begin()       { return initial; }

     // terminal:

        constexpr size_type size() const { return terminal; }
        constexpr void set_size(size_ctype n) { terminal = n; }
};
```

```
template<typename Type, typename SizeType, SizeType Size>
class vector_model
{
   public:
          // type aliases go here.
   protected:

     type initial[Size]; // compile time compatible.
     size_type terminal; // compile time compatible.

   public:
     constexpr vector_model() : initial{}, terminal{} { }

     // initial:

        constexpr ctype_ptr cbegin() const { return initial; }
        constexpr  type_ptr  begin()       { return initial; }

     // terminal:

        constexpr size_type size() const { return terminal; }
        constexpr void set_size(size_ctype n) { terminal = n; }
};
```

```
template<typename Type, typename SizeType, SizeType Size>
class vector_model
{
   public:
          // type aliases go here.
   protected:

     type initial[Size]; // compile time compatible.
     size_type terminal; // compile time compatible.

   public:
     constexpr vector_model() : initial{}, terminal{} { }

     // initial:

        constexpr ctype_ptr cbegin() const { return initial; }
        constexpr  type_ptr  begin()       { return initial; }

     // terminal:

        constexpr size_type size() const { return terminal; }
        constexpr void set_size(size_ctype n) { terminal = n; }
};
```

Next in our paradigm:

Next in our paradigm: The facade.

Intuitively our lens classes maintain pointers to memory models, Intuitively our lens classes maintain pointers to memory models, but for the sake of refactoring (among other things)

Intuitively our lens classes maintain pointers to memory models, but for the sake of refactoring (among other things) we add facades as an extra level of indirection.

Facades hold pointers to models

Facades hold pointers to models instead of the lenses.

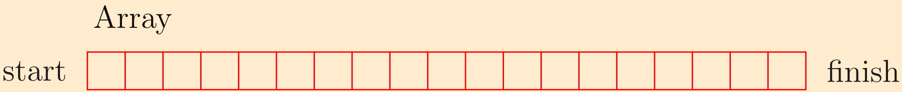
Facades hold pointers to models instead of the lenses.
Lenses inherit from facades.

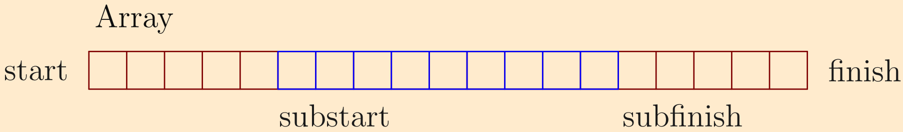
```
template<typename Model>
class vector_facade
{
   public:
         // type aliases go here.
   protected:

      model_type_ptr model;

   public:
      constexpr vector_facade() { }
      constexpr vector_facade(model_type_cptr m) : model{m} { }

      // initial:

         constexpr ctype_ptr cbegin() const { return model->cbegin(); }
         constexpr type_ptr begin() { return model->begin(); }

      // terminal:

         constexpr size_type size() const { return model->size(); }
         constexpr void set_size(size_ctype n) { model->set_size(n); }
};
```

```
template<typename Model>
class vector_facade
{
   public:
          // type aliases go here.
   protected:

     model_type_ptr model;

   public:
     constexpr vector_facade() { }
     constexpr vector_facade(model_type_cptr m) : model{m} { }

     // initial:

        constexpr ctype_ptr cbegin() const { return model->cbegin(); }
        constexpr type_ptr begin() { return model->begin(); }

     // terminal:

        constexpr size_type size() const { return model->size(); }
        constexpr void set_size(size_ctype n) { model->set_size(n); }
};
```

```
template<typename Model>
class vector_facade
{
   public:
          // type aliases go here.
   protected:

     model_type_ptr model;

   public:
     constexpr vector_facade() { }
     constexpr vector_facade(model_type_cptr m) : model{m} { }

     // initial:

        constexpr ctype_ptr cbegin() const { return model->cbegin(); }
        constexpr type_ptr begin() { return model->begin(); }

     // terminal:

        constexpr size_type size() const { return model->size(); }
        constexpr void set_size(size_ctype n) { model->set_size(n); }
};
```

The second major reason for introducing facade classes is to promote subcontainer interfaces:


Array
start
finish



Array
start
finish
substart
subfinish

We define an interval subfacade here:

```
template<typename Model>
class mutable_interval_facade
{
   public:
          // type aliases go here.
   protected:

     model_type_ptr model;
     size_type  initial;
     size_type terminal;

   public:
     constexpr mutable_interval_facade() { }
     constexpr mutable_interval_facade(model_type_cptr m) :
       model{m}, initial{}, terminal{} { }

     // initial:

        constexpr type_ptr begin() { return model->begin() + initial; }
        // class methods go here.

     // terminal:

        constexpr void set_size(size_ctype n) { terminal = n; }
        // class methods go here.
};
```

```
template<typename Model>
class mutable_interval_facade
{
   public:
          // type aliases go here.
   protected:

     model_type_ptr model;
     size_type  initial;
     size_type terminal;

   public:
     constexpr mutable_interval_facade() { }
     constexpr mutable_interval_facade(model_type_cptr m) :
       model{m}, initial{}, terminal{} { }

     // initial:

        constexpr type_ptr begin() { return model->begin() + initial; }
        // class methods go here.

     // terminal:

        constexpr void set_size(size_ctype n) { terminal = n; }
        // class methods go here.
};
```

```
template<typename Model>
class mutable_interval_facade
{
   public:
          // type aliases go here.
   protected:

     model_type_ptr model;
     size_type  initial;
     size_type terminal;

   public:
     constexpr mutable_interval_facade() { }
     constexpr mutable_interval_facade(model_type_cptr m) :
       model{m}, initial{}, terminal{} { }

     // initial:

        constexpr type_ptr begin() { return model->begin() + initial; }
        // class methods go here.

     // terminal:

        constexpr void set_size(size_ctype n) { terminal = n; }
        // class methods go here.
};
```

The final relationship in our paradigm

The final relationship in our paradigm comes from the lens classes.

To complete our implementation toolset

To complete our implementation toolset I need to introduce another paradigm called:

To complete our implementation toolset I need to introduce another paradigm called: disjoint single inheritance.

As the number of lens classes grow, we will want to refactor.

As the number of lens classes grow, we will want to refactor.

Mitigating such complexity is the purpose of disjoint single inheritance.

The idea is that any given lens definition starts with a facade from which it inherits.

There will be many lenses which are semi-related, There will be many lenses which are semi-related, and can thus be refactored, There will be many lenses which are semi-related, and can thus be refactored, but in doing so

There will be many lenses which are semi-related, and can thus be refactored, but in doing so we want such factored classes to be reusable.

In theory we could also make use of multiple inheritance, In theory we could also make use of multiple inheritance, but that brings in new complexities.

In theory we could also make use of multiple inheritance, but that brings in new complexities.

I won't get into it, but I decided no.

As for single inheritance,

As for single inheritance, inheritance itself introduces potential new problems, As for single inheritance, inheritance itself introduces potential new problems, notably:

As for single inheritance, inheritance itself introduces potential new problems, notably: Name collisions.

This is the meaning behind disjoint single inheritance.

This is the meaning behind disjoint single inheritance. Whatever classes we bind together through a chain of single inheritances, This is the meaning behind disjoint single inheritance. Whatever classes we bind together through a chain of single inheritances, it's required that they each have disjoint method names.

As for how we implement this paradigm,

As for how we implement this paradigm, we use inheritance as our natural composition mechanism.

As for how we implement this paradigm, we use inheritance as our natural composition mechanism.

The exception to this is aliases as they don't inherit directly.

As for how we implement this paradigm, we use inheritance as our natural composition mechanism.

The exception to this is aliases as they don't inherit directly. Currently my approach is to redeclare all aliases in each inherited class.

```
template<typename Base>
class strlit_csublens_disjoint : public Base
{
   public:
         // type aliases go here.
   public:
     constexpr strlit_csublens_disjoint() : base{} { }
     constexpr strlit_csublens_disjoint(const facade & f) :
       base{f} { }

     // initial:

        constexpr citer_type citer(size_ctype n) const
          { return base::cbegin() + n; }

        constexpr cderef_ctype_ref cat(size_ctype n) const
          { return *citer(n); }

     // terminal:

        constexpr citer_type cend() const
          { return citer(base::size()); }
};
```

```
template<typename Base>
class strlit_csublens_disjoint : public Base
{
   public:
         // type aliases go here.
   public:
     constexpr strlit_csublens_disjoint() : base{} { }
     constexpr strlit_csublens_disjoint(const facade & f) :
       base{f} { }

     // initial:

        constexpr citer_type citer(size_ctype n) const
          { return base::cbegin() + n; }

        constexpr cderef_ctype_ref cat(size_ctype n) const
          { return *citer(n); }

     // terminal:

        constexpr citer_type cend() const
          { return citer(base::size()); }
};
```

```
template<typename Base>
class strlit_csublens_disjoint : public Base
{
   public:
         // type aliases go here.
   public:
     constexpr strlit_csublens_disjoint() : base{} { }
     constexpr strlit_csublens_disjoint(const facade & f) :
       base{f} { }

     // initial:

        constexpr citer_type citer(size_ctype n) const
          { return base::cbegin() + n; }

        constexpr cderef_ctype_ref cat(size_ctype n) const
          { return *citer(n); }

     // terminal:

        constexpr citer_type cend() const
          { return citer(base::size()); }
};
```

```cpp
template<typename Base>
class strlit_csublens_disjoint : public Base
{
   public:
         // type aliases go here.
   public:
     constexpr strlit_csublens_disjoint() : base{} { }
     constexpr strlit_csublens_disjoint(const facade & f) :
       base{f} { }

     // initial:

        constexpr citer_type citer(size_ctype n) const
          { return base::cbegin() + n; }

        constexpr cderef_ctype_ref cat(size_ctype n) const
          { return *citer(n); }

     // terminal:

        constexpr citer_type cend() const
          { return citer(base::size()); }
};
```

Once we have enough of these disjoint lens classes, Once we have enough of these disjoint lens classes, we compose them together to build our complete ones:

```
// syntactic sugar:

   template<typename Facade>
   using vector_lens =
     vector_lens_disjoint     <
     array_sublens_disjoint   <
     array_csublens_disjoint  < // copy
     strlit_clens_disjoint    < // origin
     strlit_csublens_disjoint < Facade >>>>>;
```

```
// syntactic sugar:

  template<typename Facade>
  using vector_lens =
    vector_lens_disjoint      <
    array_sublens_disjoint    <
    array_csublens_disjoint   < // copy
    strlit_clens_disjoint     < // origin
    strlit_csublens_disjoint < Facade >>>>>;
```

Proof Assistant

We now have some *potent* and *expressive* compile time paradigms available to us.

The structures they afford are a good fit to solve many computational problems, The structures they afford are a good fit to solve many computational problems, but our goal is to build a universal structure so we can solve any problem.

The universal structure I have chosen to build is a proof assistant.

The universal structure I have chosen to build is a proof assistant.

It is also known as a theorem prover.

The universal structure I have chosen to build is a proof assistant.

It is also known as a theorem prover. Interactive theorem provers already exist in programming languages such as Rocq, Lean, Idris, Isabelle.

Our proof assistant will not be interactive.

Our proof assistant will not be interactive.
It is based off of formal logic,

Our proof assistant will not be interactive.
It is based off of formal logic,
specifically type theory.

I take my theoretical designs from two sources:

I take my theoretical designs from two sources:

- Type Theory and Functional Programming

I take my theoretical designs from two sources:

- Type Theory and Functional Programming
  by Simon Thompson.

I take my theoretical designs from two sources:

- Type Theory and Functional Programming by Simon Thompson.
- Homotopy Type Theory

I take my theoretical designs from two sources:

- Type Theory and Functional Programming
  by Simon Thompson.
- Homotopy Type Theory
  a collaborative effort by
  the Institute for Advanced Study.

Why build a proof assistant?

What are proofs?

What are proofs? The simple answer:

What are proofs? The simple answer:

Proofs are evidence that we've verified the truth of some logical claim.

In C++ we have concepts.

In C++ we have concepts.

We have requirements and satisifiability conditions

In C++ we have concepts.

We have requirements and satisifiability conditions which the compiler checks for us.

In C++ we have concepts.

We have requirements and satisifiability conditions which the compiler checks for us.

To an extent,

In C++ we have concepts.

We have requirements and satisifiability conditions which the compiler checks for us.

To an extent, when we write concept definitions we're already writing proofs.

My thinking is:

My thinking is:

If we're gonna build a universal data structure,

My thinking is:

If we're gonna build a universal data structure, we might as well build one

My thinking is:

If we're gonna build a universal data structure, we might as well build one which can also verify truths about our data

My thinking is:

If we're gonna build a universal data structure, we might as well build one which can also verify truths about our data and its structure.

The other reason for choosing this style of proof assistant

The other reason for choosing this style of proof assistant is because it is based on type theory.

In C++ we have a type system to validate our code.

In C++ we have a type system to validate our code.

A proof assistant based on type theory means we can validate our code

In C++ we have a type system to validate our code.

A proof assistant based on type theory means we can validate our code, and we can verify it beyond what C++ currently allows.

As for proofs,

As for proofs,

the word proof itself might sound scary to some,

As for proofs,

the word proof itself might sound scary to some,
so let's discuss it a bit.

If we're building a proof assistant,

If we're building a proof assistant,
it means we're dealing with math proofs.

This isn't a math conference though,

This isn't a math conference though,
so I won't go heavy into the logic.

This isn't a math conference though,
so I won't go heavy into the logic.

I aim to show here

This isn't a math conference though,
so I won't go heavy into the logic.

I aim to show here the sorts of proofs programmers will want to build

This isn't a math conference though,
so I won't go heavy into the logic.

I aim to show here the sorts of proofs programmers will want to build aren't much different than the ideas you've already been working with

This isn't a math conference though,
so I won't go heavy into the logic.

I aim to show here the sorts of proofs programmers will want to build aren't much different than the ideas you've already been working with in languages like C++.

We start with an intuitive introduction.

We start with an intuitive introduction.

When we think of math proofs

We start with an intuitive introduction.

When we think of math proofs we think of something like this:

Theorem: -(-5) = 5

Theorem: -(-5) $=$ 5

(the negative of negative five is five)

Proof:

Proof:

By definition, (-5) $+ 5 = 0$.

But if (-5) is a number all on its own,

But if (-5) is a number all on its own,

then: $(\text{-}(\text{-}5)) + (\text{-}5) = 0$.

Now:

Now:

-(-5)

Now:

$$
\begin{aligned}
& \text{-(-5)} \\
= \; & \text{(-(-5))}
\end{aligned}
$$

Now:

$$
\begin{aligned}
& \text{-(-5)} \\
&= (\text{-}(\text{-}5)) \\
&= (\text{-}(\text{-}5)) + 0
\end{aligned}
$$

Now:

$$
\begin{aligned}
& \text{-}(\text{-}5) \\
= {} & (\text{-}(\text{-}5)) \\
= {} & (\text{-}(\text{-}5)) + 0 \\
= {} & (\text{-}(\text{-}5)) + ((\text{-}5) + 5)
\end{aligned}
$$

Now:

$$
\begin{aligned}
& \text{-}(\text{-}5) \\
= {} & (\text{-}(\text{-}5)) \\
= {} & (\text{-}(\text{-}5)) + 0 \\
= {} & (\text{-}(\text{-}5)) + ((\text{-}5) + 5) \\
= {} & ((\text{-}(\text{-}5)) + (\text{-}5)) + 5
\end{aligned}
$$

Now:

$$
\begin{aligned}
&\text{-(-5)} \\
&= (\text{-}(\text{-}5)) \\
&= (\text{-}(\text{-}5)) + 0 \\
&= (\text{-}(\text{-}5)) + ((\text{-}5) + 5) \\
&= ((\text{-}(\text{-}5)) + (\text{-}5)) + 5 \\
&= 0 + 5
\end{aligned}
$$

Now:

$$
\begin{aligned}
& \text{-}(\text{-}5) \\
&= (\text{-}(\text{-}5)) \\
&= (\text{-}(\text{-}5)) + 0 \\
&= (\text{-}(\text{-}5)) + ((\text{-}5) + 5) \\
&= ((\text{-}(\text{-}5)) + (\text{-}5)) + 5 \\
&= 0 + 5 \\
&= 5
\end{aligned}
$$

So that's a proof,

So that's a proof,

but how do we write it as computer code?

So that's a proof,

but how do we write it as computer code?
How do we represent a proof in memory?

We could write it as is

We could write it as is — as string literals.

We could write it as is — as string literals.

After all,

We could write it as is — as string literals.

After all, math notation has been around for like,

We could write it as is — as string literals.

After all, math notation has been around for like, ever.

We could write it as is — as string literals.

After all, math notation has been around for like, ever.
It's pretty stable.

Then again,

Then again, our verification system would likely take a performance hit

Then again, our verification system would likely take a performance hit having to parse what is meant to be human readable proofs.

We could instead come up with our own internal representation for proofs, We could instead come up with our own internal representation for proofs, such as the "temporal logic of actions" (TLA+)

We could instead come up with our own internal representation for proofs, such as the "temporal logic of actions" (TLA+)
by Leslie Lamport.

Instead we're taking the approach based on what's known as the Curry-Howard correspondence.

This type theoretic approach recognizes a strong similarity

This type theoretic approach recognizes a strong similarity between type systems and proof systems, This type theoretic approach recognizes a strong similarity between type systems and proof systems, and in effect refactors them into one.

This type theoretic approach recognizes a strong similarity between type systems and proof systems, and in effect refactors them into one.

This will make more sense with some examples.

In type theory you start with atomic types:

In type theory you start with atomic types:

$$a : A$$

In C++ your atomics would most naturally be the builtin integer types,

In C++ your atomics would most naturally be the builtin integer types, for example:

In C++ your atomics would most naturally be the builtin integer types, for example:

$$17 : int$$

In C++ your atomics would most naturally be the builtin integer types, for example:

$$17 : int$$

(using type theory notation)

From atomics we then build compounds.

From atomics we then build compounds.
The most basic compound type is a product:

From atomics we then build compounds.
The most basic compound type is a product:

$$A \times B$$

In C++,

In C++, and to keep things simple,

In C++, and to keep things simple,
this equates with 2-tuples, or the pair type.

In functional programming, the second most basic compound type is the coproduct:

In functional programming, the second most basic compound type is the coproduct:

$$A \sqcup B$$

In C++ this equates with union or variant types.

In functional programming, all data structures built out of atomic, product, and coproduct types are generally called algebraic data types.

From there, we introduce recursive data types.

From there, we introduce recursive data types.
For this talk I'll just mention list types:

From there, we introduce recursive data types.
For this talk I'll just mention list types:

$$\mathrm{List}_A \quad := \quad \emptyset_A \sqcup (A \times \mathrm{List}_A)$$

Here, $\emptyset_A$ is the empty list of type A.

In C++ list types equate with linked lists.

If we have algebraic and recursive types, If we have algebraic and recursive types, along with common atomics such as *boolean* and *unicode characters*, If we have algebraic and recursive types, along with common atomics such as *boolean* and *unicode characters*, we theoretically have enough to solve many computational problems in the world of programming.

I could go more into the subtleties of type theory, but what about proofs?

I could go more into the subtleties of type theory, but what about proofs?

What's the similarity with types?

In math we have theorems,

In math we have theorems, lemmas,

In math we have theorems, lemmas, corollaries, In math we have theorems, lemmas, corollaries, propositions, In math we have theorems, lemmas, corollaries, propositions, and predicates.

Let's keep things simple and call them all:

Let’s keep things simple and call them all: Claims.

So let's say we have a claim:

$$\text{Claim}[A]$$

If we can verify the claim, we have a proof for it:

$$p : \mathrm{Claim}[A]$$

What if we have two claims, each with its own proof:

$$p : \text{Claim}[A]$$
$$q : \text{Claim}[B]$$

In logic we have operators such as (and), (or).

In logic we have operators such as (and), (or).

Can we connect our two claims into a new one?

In logic we have operators such as (and), (or).

Can we connect our two claims into a new one?

$$\text{Claim}[C] \quad := \quad \text{Claim}[A] \text{ and } \text{Claim}[B]$$

If so, how do we represent the *proof* of this new claim?

The intuitive answer is to say:

The intuitive answer is to say: If Claim[A] is true and has a proof, The intuitive answer is to say: If Claim[A] is true and has a proof, and if Claim[B] is true and has a proof,

The intuitive answer is to say: If Claim[A] is true and has a proof, and if Claim[B] is true and has a proof, then the proof of their conjunction:

The intuitive answer is to say: If Claim[A] is true and has a proof, and if Claim[B] is true and has a proof, then the proof of their conjunction:

$$\text{Claim}[A] \text{ and } \text{Claim}[B]$$

Is just the two proofs bound together.

The intuitive answer is to say: If Claim[A] is true and has a proof, and if Claim[B] is true and has a proof, then the proof of their conjunction:

$$\text{Claim}[A] \text{ and } \text{Claim}[B]$$

Is just the two proofs bound together. A natural way to bind these proofs is with a pair:

$$(p, q) : \text{Claim}[A] \text{ and } \text{Claim}[B]$$

It's similar with the (or) operator, but there's a twist.

It's similar with the (or) operator, but there's a twist.
A proof of the disjunction:

$$\text{Claim}[A] \text{ or } \text{Claim}[B]$$

It's similar with the (or) operator, but there's a twist.
A proof of the disjunction:

$$\mathrm{Claim}[A] \text{ or } \mathrm{Claim}[B]$$

only requires a single proof of
either $\mathrm{Claim}[A]$ or of $\mathrm{Claim}[B]$.

Keep in mind both claims could have proofs, but for us we're only interested in knowing at least one does.

The thing is, we still need to represent this proof in computer memory, which means we have to make explicit the proof we're using.

$$p : \mathrm{Claim}[A] \text{ or } \mathrm{Claim}[B]$$

Here we run into a problem.

Here we run into a problem.

We've attached a proof to this compound claim,

Here we run into a problem.

We've attached a proof to this compound claim, but it's unclear which subclaim it comes from.

Here we run into a problem.

We've attached a proof to this compound claim, but it's unclear which subclaim it comes from. Does this matter?

Here we run into a problem.

We've attached a proof to this compound claim, but it's unclear which subclaim it comes from. Does this matter?

Sometimes yes,

Here we run into a problem.

We've attached a proof to this compound claim, but it's unclear which subclaim it comes from. Does this matter?

Sometimes yes, sometimes no,

Here we run into a problem.

We've attached a proof to this compound claim, but it's unclear which subclaim it comes from. Does this matter?

Sometimes yes, sometimes no, but to keep our design simple

Here we run into a problem.

We've attached a proof to this compound claim, but it's unclear which subclaim it comes from. Does this matter?

Sometimes yes, sometimes no, but to keep our design simple it's best just to *wrap* our proof with where it came from:

$$\text{from}_A(p) : \text{Claim}[A] \text{ or } \text{Claim}[B]$$

How is this similar to type theory?

This coincides with the coproduct type.

This coincides with the coproduct type.

Let's say we want a variant of:

(C++ conferences) $\sqcup$ (2025 conferences)

In this case, an object of this type is:

CppNorth : (C++ conferences) $\sqcup$ (2025 conferences)

What if we wanted to know which type this CppNorth value belongs too?

What if we wanted to know which type this CppNorth value belongs too?

It is an object of both subtypes, so it's ambiguous.

For recordkeeping purposes then,

For recordkeeping purposes then,
our coproduct type wraps its values as well:

$\text{from}_{C++}(\text{CppNorth}) :$
$(\text{C++ conferences}) \sqcup (\text{2025 conferences})$

There is one more logical connective to discuss:

$$\text{Claim}[A] \implies \text{Claim}[B]$$

How do we represent a proof of implication?

We start with a proof of the *initial* (leftside) claim:

$$p : \mathrm{Claim}[A]$$

And we map it to a proof of the *terminal* (rightside) claim:

$$q : \mathrm{Claim}[B]$$

$$p : \mathrm{Claim}[A] \quad \rightarrow \quad q : \mathrm{Claim}[B]$$

But what if our initial claim has more than one proof?

$$\{p_1,\ p_2,\ \ldots\}\ :\ \mathrm{Claim}[A]$$

We want our proof of implication
to be as neutral as possible.

We want our proof of implication
to be as neutral as possible.

We do this by showing that any proof of the initial claim

We want our proof of implication
to be as neutral as possible.

We do this by showing that any proof of the initial claim gets mapped to a proof of the terminal claim:

We want our proof of implication
to be as neutral as possible.

We do this by showing that any proof of the initial claim gets mapped to a proof of the terminal claim:

$$\text{f(p)} : \text{Claim}[A] \implies \text{Claim}[B]$$

This is to say,

This is to say, an implication claim equates with a function type, This is to say, an implication claim equates with a function type, and a proof of that claim equates with a function itself.

The thing to note about all of this

The thing to note about all of this is that for each Claim[A] format we introduce, The thing to note about all of this is that for each Claim$[A]$ format we introduce, it tends to correspond with a type and a value from type theory.

What about the list type then?

What about the list type then?

Lists are recursive,

What about the list type then?

Lists are recursive, but otherwise are defined in terms of products and coproducts.

What about the list type then?

Lists are recursive, but otherwise are defined in terms of products and coproducts. There is a corresponding proof format,

What about the list type then?

Lists are recursive, but otherwise are defined in terms of products and coproducts. There is a corresponding proof format, but it doesn't have any deep meaning in this talk.

As for the Curry-Howard correspondence,

As for the Curry-Howard correspondence,
it works both ways:

As for the Curry-Howard correspondence,
it works both ways:

Types and their values can be used to represent claims
and their proofs,

As for the Curry-Howard correspondence,
it works both ways:

Types and their values can be used to represent claims and their proofs, but claims and their proofs can inspire new types as well.

In logic we can make predicate claims such as:

$$(\forall x : A) \,.\, Claim[B(x)]$$

which reads as:

In logic we can make predicate claims such as:

$$(\forall x : A) \,.\, Claim[B(x)]$$

which reads as: For all $x : A$, our claim holds for $B(x)$.

In logic we can make predicate claims such as:

$$(\forall x : A) \,.\, Claim[B(x)]$$

which reads as: For all $x : A$, our claim holds for $B(x)$.

This predicate is known as a universal quantifier.

How do we represent a proof such as this?

$$p : (\forall x : A)\,.\,Claim[B(x)]$$

How do we represent a proof such as this?

$$p : (\forall x : A) \,.\, Claim[B(x)]$$

And what is the corresponding type?

In this case, we need separate proofs for each value x, which is to say:

$$p(x) : (\forall x : A) \,.\, Claim[B(x)]$$

In this case, we need separate proofs for each value x, which is to say:

$$p(x) : (\forall x : A) \,.\, Claim[B(x)]$$

Our proof is a function.

In this instance though, our function is a bit different than the one we encountered with implication.

Here logic inspires type theory.

Here logic inspires type theory.

The type corresponding to this predicate claim is called a dependent function type,

Here logic inspires type theory.

The type corresponding to this predicate claim is called a dependent function type, the value being a dependent function.

Think of it like this:

$$f(x) : A \to B(x)$$

That is, we have a function but with a twist:

That is, we have a function but with a twist:

To evaluate this function we first take the input x,

That is, we have a function but with a twist:

To evaluate this function we first take the input x, and use it to determine the output type $B(x)$.

That is, we have a function but with a twist:

To evaluate this function we first take the input x, and use it to determine the output type $B(x)$.

Once that's resolved,

That is, we have a function but with a twist:

To evaluate this function we first take the input x, and use it to determine the output type $B(x)$.

Once that's resolved, we can then evaluate it like the functions we're use to.

As for the predicate claim:

$$p(x) : (\forall x : A) \,.\, Claim[B(x)]$$

Our notation might appear a bit different,

Our notation might appear a bit different,
but the proof is in fact a dependent function.

We start with the input $x : A$,

We start with the input $x : A$, use it to resolve the output claim $B(x)$, We start with the input $x : A$, use it to resolve the output claim $B(x)$, then we map the proof value x to a proof p(x):Claim[B(x)], We start with the input $x : A$, use it to resolve the output claim $B(x)$, then we map the proof value x to a proof p(x):Claim[B(x)], and we do this for each x.

If at all you're thinking this is abstract, If at all you're thinking this is abstract, not practical, If at all you're thinking this is abstract, not practical, I will add that this sort of type shows up in C++ as well.

The simplest example is the array type:

```
std::array<int, 5>
std::array<int, 6>
std::array<int, 7>
```

The function for this type is an array constructor.

The function for this type is an array constructor.
For each input variable n, we construct the array type:

$f(n)$: int $\rightarrow$ std::array<int, n>

The function for this type is an array constructor.
For each input variable n, we construct the array type:

$f(n) : \text{int} \to \text{std::array<int, n>}$

We then use that input n to initialize a value:

$f(n) := \text{std::array<int, n>}\{\}$

The other logical predicate worth mentioning is the existential quantifier:

$$(\exists x : A) \,.\, \mathit{Claim}[B(x)]$$

it reads: There exists $x : A$, such that our claim holds for $B(x)$.

In this case, we represent a proof using what's called a dependent pair:

$$(x, p(x)) : (\exists x : A) \,.\, Claim[B(x)]$$

This is to say,

This is to say, we only need to demonstrate a single value $x : A$

This is to say, we only need to demonstrate a single value $x : A$ and a single proof $p(x)$ satisfying $B(x)$

This is to say, we only need to demonstrate a single value $x : A$ and a single proof $p(x)$ satisfying $B(x)$ to represent a proof of the larger predicate claim.

The type corresponding to this style of claim is called a dependent pair type.

And we're done.

And we're done.

Generally speaking,

And we're done.

Generally speaking, that's largely what we need to know about Curry-Howard type systems

And we're done.

Generally speaking, that's largely what we need to know about Curry-Howard type systems to continue with our proof assistant.

We're nearly ready for some applications

We're nearly ready for some applications, but first I thought I'd summarize a major difference with regular C++ programming.

In practice, our logical claims are generally comparisons such as equality:

$$p : (a ==_C b)$$

In practice, our logical claims are generally comparisons such as equality:

$$p : (a ==_C b)$$

This says that $a, b : C$ and that we have a proof p of this claim.

To put this another way,

To put this another way, $(a ==_C b)$ is a type,

To put this another way, $(a ==_C b)$ is a type,
not a "true or false statement" like you're use to in C++.

It's a subtle difference.

It’s a subtle difference.

The idea is,

It's a subtle difference.

The idea is, we can view this type as *true* when it is inhabited,

It's a subtle difference.

The idea is, we can view this type as *true* when it is inhabited, which is to say:

It's a subtle difference.

The idea is, we can view this type as *true* when it is inhabited, which is to say: It is true if it has a proof.

Finally,

Finally,
a note about proof assistants more generally:

Finally,
a note about proof assistants more generally:

I've covered most of the introductory ideas here, but there are a few subtleties I haven't.

Finally,
a note about proof assistants more generally:

I've covered most of the introductory ideas here, but there are a few subtleties I haven't. Notably the definition of logical negation:

$$\neg A \quad := \quad A \to \emptyset$$

Although I leave it to the audience here,

Although I leave it to the audience here,

I will say that this definition of negation leads to different understandings

Although I leave it to the audience here,

I will say that this definition of negation leads to different understandings of the law of excluded middle

Although I leave it to the audience here,

I will say that this definition of negation leads to different understandings of the law of excluded middle as well as ideas of proof by contradiction.

Applications

The main application here

The main application here is to implement a proof assistant

The main application here is to implement a proof assistant which is meant to be our universal data structure.

We implement it

We implement it using the method equip paradigm

We implement it using the method equip paradigm along with continuant machines, We implement it using the method equip paradigm along with continuant machines, both applied to compile time (inplace) vectors.

In my library I call such proof assistant classes concords.

These concord classes hold types and claims, These concord classes hold types and claims, but are ultimately meant to hold values and proofs.

These concord classes hold types and claims, but are ultimately meant to hold values and proofs. Creating such values is generally a 3 step process:

These concord classes hold types and claims, but are ultimately meant to hold values and proofs. Creating such values is generally a 3 step process:

1. declare the type.

These concord classes hold types and claims, but are ultimately meant to hold values and proofs. Creating such values is generally a 3 step process:

1. declare the type.
2. declare the value.

These concord classes hold types and claims, but are ultimately meant to hold values and proofs. Creating such values is generally a 3 step process:

1. declare the type.
2. declare the value.
3. define the value.

Declaring a type is relatively simple:

Declaring a type is relatively simple:

```
using concord_type      = concord<unsigned, unsigned, 1000>;
using boolean_lens_type = resolve_lens<concord_type, boolean_methods>;

concord_type concord;

auto boolean_lens       = concord.template equip<boolean_lens_type>();

auto boolean_icon       = boolean_lens.declare_type();
```

The icon object returned

The icon object returned is defined to be a class wrapped around a C++ int type.

The wrapper helps with C++ type checking, The wrapper helps with C++ type checking, and otherwise has the purpose of keeping track of where within the concord (vector)

The wrapper helps with C++ type checking, and otherwise has the purpose of keeping track of where within the concord (vector) our type info is located.

Compound types are pretty much the same, except they also take icon objects as input.

With that said,

With that said, functions (as values) are a special case on their own.

The internal representation of functions

The internal representation of functions within this type system

The internal representation of functions within this type system are defined using continuant machine assembly.

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

Rather than coders defining such functions directly, Rather than coders defining such functions directly, I've added to this type system a lambda expression domain specific language

Rather than coders defining such functions directly, I've added to this type system a lambda expression domain specific language as a way of constructing functions from scratch.

Here when I say lambda,

Here when I say lambda,
I'm not referring to C++ lambdas,

Here when I say lambda,
I'm not referring to C++ lambdas,
but rather the lambda calculus.

Here when I say lambda,
I'm not referring to C++ lambdas,
but rather the lambda calculus.

I won't go into the theory, but the code for function construction looks something like this:

```
constexpr auto make_sum_of_squares(lambdex_type & lambdex) // lambda expressions.
{
   auto add            = lambdex.new_addition       (some_type, some_type, some_type);
   auto mult           = lambdex.new_multiplication (some_type, some_type, some_type);

   auto x              = lambdex.new_variable                 (some_type   );
   auto y              = lambdex.new_variable_different_than (some_type, x);

   auto mult_x_x       = lambdex.new_application (mult_x, x, x);
   auto square         = lambdex.new_abstraction (x, mult_x_x);

   auto sq_x           = lambdex.new_application (square, x);
   auto sq_y           = lambdex.new_application (square, y);
   auto add_sq_x_sq_y  = lambdex.new_application (add, sq_x, sq_y);
   auto sum_of_squares = lambdex.new_abstraction (x, y, add_sq_x_sq_y);

   return sum_of_squares;
}
```

As for declaring a value, it's straightforward.

As for declaring a value, it's straightforward.
We only need pass the icon to the declaration method:

As for declaring a value, it's straightforward.
We only need pass the icon to the declaration method:

```
auto boolean_sign0 = boolean_lens.
                       declare_value(boolean_icon);
```

To distinguish concord value entries from type entries, To distinguish concord value entries from type entries, we return a sign wrapper

To distinguish concord value entries from type entries, we return a sign wrapper in contrast to the previous icon wrapper.

Finally, we define values:

Finally, we define values:

```
boolean_lens.define_value(boolean_sign0, true);
```

Moving forward,

Moving forward,

There is another component to our type system design worth mentioning:

Moving forward,

There is another component to our type system design worth mentioning: Serialization.

Designing these concord classes to be serializable

Designing these concord classes to be serializable has one major purpose here:

Designing these concord classes to be serializable has one major purpose here:

It mitigates compile times.

Designing these concord classes to be serializable has one major purpose here:

It mitigates compile times. It does this in two ways.

First, the internal representation as an inplace vector is easy to write out to file,

First, the internal representation as an inplace vector is easy to write out to file, and equally as easy to read back in as a constexpr object.

First, the internal representation as an inplace vector is easy to write out to file, and equally as easy to read back in as a constexpr object.

This means we can in effect *save* our compile time work,

First, the internal representation as an inplace vector is easy to write out to file, and equally as easy to read back in as a constexpr object.

This means we can in effect *save* our compile time work, and *restore* it next time we recompile.

As for the second mitigation:

As for the second mitigation:

The internal representation as an inplace vector

As for the second mitigation:

The internal representation as an inplace vector
is naturally interoperable with continuant machines,

As for the second mitigation:

The internal representation as an inplace vector is naturally interoperable with continuant machines, meaning concord values can also be defined using constexpr functions.

This lightens the concord memory load

This lightens the concord memory load by not having to store intermediate computational values

This lightens the concord memory load by not having to store intermediate computational values as we build up other values in our system.

I wish I could give greater detail here,

I wish I could give greater detail here,
but that is currently it for our proof assistant.

I wish I could give greater detail here,
but that is currently it for our proof assistant.

Proof values are introduced in the same way,

I wish I could give greater detail here,
but that is currently it for our proof assistant.

Proof values are introduced in the same way,
except atomic proofs are created
using specific builtin methods.

As for real world applications of such a proof assistant?

As for real world applications of such a proof assistant?

C++ safety among other things.

For example:

For example:

We can define at metacompile time

For example:

We can define at metacompile time
a concord object which holds a handful
of constant values

For example:

We can define at metacompile time
a concord object which holds a handful
of constant values as well as
an inventory of functions.

Because we've defined these at metacompile time, Because we've defined these at metacompile time, once constructed the concord object becomes a proper *compile time* object, Because we've defined these at metacompile time, once constructed the concord object becomes a proper *compile time* object, meaning the continuant assembly representing functions internally are also compile time objects

Because we've defined these at metacompile time, once constructed the concord object becomes a proper *compile time* object, meaning the continuant assembly representing functions internally are also compile time objects which can be translated into C++ constexpr functions

Because we've defined these at metacompile time, once constructed the concord object becomes a proper *compile time* object, meaning the continuant assembly representing functions internally are also compile time objects which can be translated into C++ constexpr functions using the *metacompiler paradigm*.

We then reference and use these compile time concords

We then reference and use these compile time concords within the definitions of regular templated C++ classes.

We then reference and use these compile time concords within the definitions of regular templated C++ classes.

In a generic sense,

We then reference and use these compile time concords within the definitions of regular templated C++ classes.

In a generic sense, this much alone is already quite similar to the object oriented paradigm, where we define classes to have constant member values as well as an inventory of methods.

The difference is we can write specifications about our C++ classes and formally prove them.

The difference is we can write specifications about our C++ classes and formally prove them.

This even goes beyond what C++ concepts are capable of verifying.

In time,

In time, if I might be so bold:

In time, if I might be so bold:

We could even rewrite parts of a C++ compiler itself,

In time, if I might be so bold:

We could even rewrite parts of a C++ compiler itself, proving we've met its specification requirements.

End

(thank you)

References

- Daniel Nikpayuk, C++ is a Metacompiler, 2024: https://www.youtube.com/watch?v=zngToaBjHVk
- non-transient constexpr allocation: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2670r0.html
- What's so hard about constexpr allocation? https://brevzin.github.io/c++/2024/07/24/constexpr-alloc/
- Type Theory and Functional Programming: https://www.cs.kent.ac.uk/people/staff/sjt/TTFP/ttfp.pdf
- Homotopy Type Theory: https://homotopytypetheory.org/book/