# Generic implementation strategies in Carbon and Clang

**Richard Smith**

@zygoloid

Carbon / Google

LLVM Developers' Meeting 2024

# C++ templates and Carbon generics

C++

```cpp
template<typename T> T clamp_nonnegative(const T &a) {
  return a < T() ? T() : a;
}
```

# C++ templates and Carbon generics

```
template<typename T> T clamp_nonnegative(const T &a) {
  return a < T() ? T() : a;
}
```

C++

- Not fully type-checkable when defined
- Substitution into template produces a *template instantiation*
- Instantiation of function template is a normal function

# C++ templates and Carbon generics

Carbon

```
fn ClampNonnegative[template T:! type](a: T) -> T {
  return if a < (0 as T) then 0 as T else a;
}
```

# C++ templates and Carbon generics

```
fn ClampNonnegative[template T:! type](a: T) -> T {
  return if a < (0 as T) then 0 as T else a;
}
```

Carbon

- Not fully type-checkable when defined
- Substitution into generic produces a *specific*
- Specific for generic function is a normal function

# C++ templates and Carbon generics

```
fn ClampNonnegative[template T:! type](a: T) -> T {
  return if a < (0 as T) then 0 as T else a;
}
```

Carbon

- Not fully type-checkable when defined
- Substitution into generic produces a *specific*
- Specific for generic function is a normal function

```
fn ClampNonnegative[T:! Numeric & Ordered](a: T) -> T {
  return if a < (0 as T) then 0 as T else a;
}
```

Carbon

- Non-template generics are fully type-checked when they are defined

# C++ template representations

```cpp
template<typename T> T clamp_nonnegative(const T &a) {
  return a < T() ? T() : a;
}
```

C++

# C++ template representations

```cpp
template<typename T> T clamp_nonnegative(const T &a) {
  return a < T() ? T() : a;
}
```

Two main approaches:

- Token soup with token replay: used by EDG, MSVC (old parser)
- Dependent parse trees with tree transform: used by Clang, GCC

# Dependent parse trees

# Dependent parse trees

Parse: build normal IR

```
template<typename T> T clamp_nonnegative(const T &a) {
  return a < T() ? T() : a;
}
```

C++

# Dependent parse trees

Parse: build normal IR

```
template<typename T> T clamp_nonnegative(const T &a) {
  return a < T() ? T() : a;
}
```

```
FunctionDecl <col:22, line:3:1> line:1:24 clamp_nonnegative 'T (const T &)'
├─ParmVarDecl <col:30, col:39> col:39 referenced a 'const T &'
└─CompoundStmt <col:42, line:3:1>
  └─ReturnStmt <line:2:3, col:26>
    └─ConditionalOperator <col:10, col:26> '<dependent type>'
      ├─BinaryOperator <col:10, col:16> '<dependent type>' '<'
      │ ├─DeclRefExpr <col:10> 'const T' lvalue ParmVar 0xd67ffd0 'a' 'const T &'
      │ └─CXXUnresolvedConstructExpr <col:14, col:16> 'T' 'T'
      ├─CXXUnresolvedConstructExpr <col:20, col:22> 'T' 'T'
      └─DeclRefExpr <col:26> 'const T' lvalue ParmVar 0xd67ffd0 'a' 'const T &'
```

… with explicit representation for dependent constructs

# Dependent parse trees

Parse: build normal IR

```cpp
template<typename T> T clamp_nonnegative(const T &a) {
  return a < T() ? T() : a;
}
```

C++

```
FunctionDecl <col:22, line:3:1> line:1:24 clamp_nonnegative 'T (const T &)'
├─ParmVarDecl <col:30, col:39> col:39 referenced a 'const T &'
└─CompoundStmt <col:42, line:3:1>
  └─ReturnStmt <line:2:3, col:26>
    └─ConditionalOperator <col:10, col:26> '<dependent type>'
      ├─BinaryOperator <col:10, col:16> '<dependent type>' '<'
      │ ├─DeclRefExpr <col:10> 'const T' lvalue ParmVar 0xd67ffd0 'a' 'const T &'
      │ └─CXXUnresolvedConstructExpr <col:14, col:16> 'T' 'T'
      ├─CXXUnresolvedConstructExpr <col:20, col:22> 'T' 'T'
      └─DeclRefExpr <col:26> 'const T' lvalue ParmVar 0xd67ffd0 'a' 'const T &'
```

C++

… with explicit representation for dependent constructs

# Dependent parse trees

Parse: build normal IR

```cpp
template<typename T> T clamp_nonnegative(const T &a) {
  return a < T() ? T() : a;
}
```

```cpp
FunctionDecl <col:22, line:3:1> line:1:24 clamp_nonnegative 'T (const T &)'
├─ParmVarDecl <col:30, col:39> col:39 referenced a 'const T &'
└─CompoundStmt <col:42, line:3:1>
  └─ReturnStmt <line:2:3, col:26>
    └─ConditionalOperator <col:10, col:26> '<dependent type>'
      ├─BinaryOperator <col:10, col:16> '<dependent type>' '<'
      │ ├─DeclRefExpr <col:10> 'const T' lvalue ParmVar 0xd67ffd0 'a' 'const T &'
      │ └─CXXUnresolvedConstructExpr <col:14, col:16> 'T' 'T'
      ├─CXXUnresolvedConstructExpr <col:20, col:22> 'T' 'T'
      └─DeclRefExpr <col:26> 'const T' lvalue ParmVar 0xd67ffd0 'a' 'const T &'
```

… with explicit representation for dependent constructs

# Dependent parse trees

Instantiate: tree transformation builds a new tree

```cpp
template<> int clamp_nonnegative<int>(const int &a) {
  return a < int() ? int() : a;
}
```

```
FunctionDecl <col:22, line:3:1> line:1:24 clamp_nonnegative 'int (const int &)'
├─ParmVarDecl <col:30, col:39> col:39 used a 'const int &'
└─CompoundStmt <col:42, line:3:1>
  └─ReturnStmt <line:2:3, col:26>
    └─ConditionalOperator <col:10, col:26> 'int'
      ├─BinaryOperator <col:10, col:16> 'bool' '<'
      │ ├─DeclRefExpr <col:10> 'const int' lvalue ParmVar 0xddc7958 'a' 'const int &'
      │ └─CXXScalarValueInitExpr <col:14, col:16> 'int'
      ├─CXXScalarValueInitExpr <col:20, col:22> 'int'
      └─DeclRefExpr <col:26> 'const int' lvalue ParmVar 0xddc7958 'a' 'const int &'
```

# Dependent parse trees

Instantiate: tree transformation builds a new tree

```cpp
template<> int clamp_nonnegative<int>(const int &a) {
  return a < int() ? int() : a;
}
```

```
FunctionDecl <col:22, line:3:1> line:1:24 clamp_nonnegative 'int (const int &)'
├─ParmVarDecl <col:30, col:39> col:39 used a 'const int &'
└─CompoundStmt <col:42, line:3:1>
  └─ReturnStmt <line:2:3, col:26>
    └─ConditionalOperator <col:10, col:26> 'int'
      ├─BinaryOperator <col:10, col:16> 'bool' '<'
      │ ├─DeclRefExpr <col:10> 'const int' lvalue ParmVar 0xddc7958 'a' 'const int &'
      │ └─CXXScalarValueInitExpr <col:14, col:16> 'int'
      ├─CXXScalarValueInitExpr <col:20, col:22> 'int'
      └─DeclRefExpr <col:26> 'const int' lvalue ParmVar 0xddc7958 'a' 'const int &'
```

# Dependent parse trees

Instantiate: tree transformation builds a new tree

```cpp
template<> int clamp_nonnegative<int>(const int &a) {
  return a < int() ? int() : a;
}
```

```
FunctionDecl <col:22, line:3:1> line:1:24 clamp_nonnegative 'int (const int &)'
├─ParmVarDecl <col:30, col:39> col:39 used a 'const int &'
└─CompoundStmt <col:42, line:3:1>
  └─ReturnStmt <line:2:3, col:26>
    └─ConditionalOperator <col:10, col:26> 'int'
      ├─BinaryOperator <col:10, col:16> 'bool' '<'
      │ ├─DeclRefExpr <col:10> 'const int' lvalue ParmVar 0xddc7958 'a' 'const int &'
      │ └─CXXScalarValueInitExpr <col:14, col:16> 'int'
      ├─CXXScalarValueInitExpr <col:20, col:22> 'int'
      └─DeclRefExpr <col:26> 'const int' lvalue ParmVar 0xddc7958 'a' 'const int &'
```

# Dependent parse trees

Instantiate: tree transformation builds a new tree

```C++
template<> int clamp_nonnegative<int>(const int &a) {
  return a < int() ? int() : a;
}
```

```C++
FunctionDecl <col:22, line:3:1> line:1:24 clamp_nonnegative 'int (const int &)'
├─ParmVarDecl <col:30, col:39> col:39 used a 'const int &'
└─CompoundStmt <col:42, line:3:1>
  └─ReturnStmt <line:2:3, col:26>
    └─ConditionalOperator <col:10, col:26> 'int'
      ├─BinaryOperator <col:10, col:16> 'bool' '<'
      │ ├─ImplicitCastExpr <col:10> 'int' <LValueToRValue>
      │ │ └─DeclRefExpr <col:10> 'const int' lvalue ParmVar 0xddc7958 'a' 'const int &'
      │ └─CXXScalarValueInitExpr <col:14, col:16> 'int'
      ├─CXXScalarValueInitExpr <col:20, col:22> 'int'
      └─ImplicitCastExpr <col:26> 'int' <LValueToRValue>
        └─DeclRefExpr <col:26> 'const int' lvalue ParmVar 0xddc7958 'a' 'const int &'
```

- May result in somewhat different tree

# Subtree reuse

## Clang `TreeTransform`

- Attempts to reuse non-dependent parts of tree

```cpp
template<range R> auto average(const R &v)
    -> range_value_t<R> {
  int n = 0;
  range_value_t<R> sum = 0;
  for (auto &elem : v) {
    sum += elem;
    ++n;
  }
  return sum / (n ? n : 1);
}
```

# Subtree reuse

Clang `TreeTransform`

- Attempts to reuse non-dependent parts of tree

```C++
template<range R> auto average(const R &v)
    -> range_value_t<R> {
  int n = 0;
  range_value_t<R> sum = 0;
  for (auto &elem : v) {
    sum += elem;
    ++n;
  }
  return sum / (n ? n : 1);
}
```

# Subtree reuse

Clang `TreeTransform`

- Attempts to reuse non-dependent parts of tree

```cpp
 1 template<range R> auto average(const R &v)
 2     -> range_value_t<R> {
 3   int n = 0;
 4   range_value_t<R> sum = 0;
 5   for (auto &elem : v) {
 6     sum += elem;
 7     ++n;
 8   }
 9   return sum / (n ? n : 1);
10 }
```

Actually reuses:

```
IntegerLiteral <line:3:11> 'int' 0
IntegerLiteral <line:4:26> 'int' 0
IntegerLiteral <line:9:25> 'int' 1
```

# Subtree reuse

Clang `TreeTransform`

- Attempts to reuse non-dependent parts of tree
- Usually fails

Why?

## Subtree reuse

Clang `TreeTransform`

- Attempts to reuse non-dependent parts of tree
- Usually fails

Why?

- Children change
- Local variables
- Types change
- Initializers
- Pack expansions

# Dependent parse trees

Cost of building instantiation

- Comparable to cost of building template
- Usually less: *some* work is shared
  - Parsing
  - Unqualified name lookup
  - Reuse some non-dependent parts of tree
- Can still be surprisingly high

# Example: type trait

C++

```cpp
template<typename T> struct is_const {
  static constexpr bool value = __is_const(T);
};
const bool b1 = is_const<int[1]>::value;
const bool b2 = is_const<int[2]>::value;
...
```

# Example: type trait

```C++
template<typename T> struct is_const {
  static constexpr bool value = __is_const(T);
};
const bool b1 = is_const<int[1]>::value;
const bool b2 = is_const<int[2]>::value;
...
```

- 43 µs, 1.6 KiB per instantiation

# Example: type trait

```cpp
template<typename T> struct is_const {
  static constexpr bool value = __is_const(T);
};
const bool b1 = is_const<int[1]>::value;
const bool b2 = is_const<int[2]>::value;
...
```

- 43 µs, 1.6 KiB per instantiation

```cpp
template<typename T>
constexpr bool is_const = __is_const(T);
const bool b1 = is_const<int[1]>;
const bool b2 = is_const<int[2]>;
...
```

# Example: type trait

```C++
template<typename T> struct is_const {
  static constexpr bool value = __is_const(T);
};
const bool b1 = is_const<int[1]>::value;
const bool b2 = is_const<int[2]>::value;
...
```

- 43 µs, 1.6 KiB per instantiation

```C++
template<typename T>
constexpr bool is_const = __is_const(T);
const bool b1 = is_const<int[1]>;
const bool b2 = is_const<int[2]>;
...
```

- 23 µs, 1.0 KiB per instantiation

# Example: type trait

C++

```cpp
template<typename T> struct is_const {
  static constexpr bool value = __is_const(T);
};
const bool b1 = is_const<int[1]>::value;
const bool b2 = is_const<int[2]>::value;
...
```

- 43 µs, 1.6 KiB per instantiation

C++

```cpp
template<typename T>
constexpr bool is_const = __is_const(T);
const bool b1 = is_const<int[1]>;
const bool b2 = is_const<int[2]>;
...
```

- 23 µs, 1.0 KiB per instantiation

Directly computing `__is_const(int[N])`:

- 0.7 µs, 0.2 KiB

# Faster? Smaller?

# Carbon approach: overlay

Idea: treat instantiation as overlay on template

- Instantiation is dependent parse tree plus set of “patches”
- Only represent the parts that change
- Only rebuild the parts that change

# Carbon toolchain

Parse generic:

- Build array of dependent constructs
- Generic refers opaquely to array elements

# Carbon toolchain

Parse generic:

- Build array of dependent constructs
- Generic refers opaquely to array elements

Build specific:

- Compute concrete value corresponding to array elements

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: R)
    -> R.Value {
  var n: i32 = 0;
  var sum: R.Value = 0;
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: R)
    -> R.Value {
  var n: i32 = 0;
  var sum: R.Value = 0;
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: R)
    -> R.Value {
  var n: i32 = 0;
  var sum: R.Value = 0;
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #0)
    -> #0.Value {
  var n: i32 = 0;
  var sum: #0.Value = 0;
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

- `#0` = `R`

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #0)
    -> #0.Value {
  var n: i32 = 0;
  var sum: #0.Value = 0;
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

- `#0` = `R`

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #0)
    -> #1 {
  var n: i32 = 0;
  var sum: #1 = 0;
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

- `#0` = `R`
- `#1` = `#0.Value`

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #0)
    -> #1 {
  var n: i32 = 0;
  var sum: #1 = 0;
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

- `#0` = `R`
- `#1` = `#0.Value`

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #0)
    -> #1 {
  var n: i32 = 0;
  var sum: #1 = 0.(ImplicitAs(#1).Convert)();
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

- `#0` = `R`
- `#1` = `#0.Value`

`R.Value impls Numeric` implies that `IntLiteral(0) impls ImplicitAs(R.Value)`

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #0)
    -> #1 {
  var n: i32 = 0;
  var sum: #1 = 0.(ImplicitAs(#1).Convert)();
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

- `#0` = `R`
- `#1` = `#0.Value`

`R.Value impls Numeric` implies that `IntLiteral(0) impls ImplicitAs(R.Value)`

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #0)
    -> #1 {
  var n: i32 = 0;
  var sum: #1 = 0.(#2)();
  for (elem in v) {
    sum += elem;
    ++n;
  }
  return sum / (if n != 0 then n else 1);
}
```

- `#0` = `R`
- `#1` = `#0.Value`
- `#2` = `(IntLiteral(0) as ImplicitAs(#1)).Convert`

# Building a generic

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #0)
    -> #1 {
  var n: i32 = 0;
  var sum: #1 = 0.(#2)();
  for (elem in v.(#3)() ... v.(#4)()) {
    sum.(#5)(elem);
    ++n;
  }
  return sum.(#6)(if n != 0 then n else 1);
}
```

- `#0` = `R`
- `#1` = `#0.Value`
- `#2` = `(IntLiteral(0) as ImplicitAs(#1)).Convert`
- `#3` = `#0.Begin`
- `#4` = `#0.End`
- `#5` = `(#1 as AddAssign).Op`
- `#6` = `(#1 as DivWith(i32)).Op`

(Pseudocode, actually done in SemIR)

# Generic representation

These are *instructions* extracted from the generic:

- `#0` = `R`
- `#1` = `#0.Value`
- `#2` = `(IntLiteral(0) as ImplicitAs(#1)).Convert`
- `#3` = `#0.Begin`
- `#4` = `#0.End`
- `#5` = `(#1 as AddAssign).Op`
- `#6` = `(#1 as DivWith(i32)).Op`

# Generic representation

These are *instructions* extracted from the generic:

- `#0` = `R`
- `#1` = `#0.Value`
- `#2` = `(IntLiteral(0) as ImplicitAs(#1)).Convert`
- `#3` = `#0.Begin`
- `#4` = `#0.End`
- `#5` = `(#1 as AddAssign).Op`
- `#6` = `(#1 as DivWith(i32)).Op`

Can represent this as a block of code

# Generic representation

We have computed a *compile-time function* to form a specific from a generic:

Carbon

```
fn MakeAverageSpecific(R:! Range where R.Value impls Numeric) -> <function> {
  let v0:! auto = R;
  let v1:! auto = v0.Value;
  let v2:! auto = (IntLiteral(0) as ImplicitAs(v1)).Convert;
  let v3:! auto = v0.Begin;
  let v4:! auto = v0.End;
  let v5:! auto = (v1 as AddAssign).Op;
  let v6:! auto = (v1 as DivWith(i32)).Op;
  return <make specific>(Average, (v0, v1, v2, v3, v4, v5, v6));
}
```

# Generic representation

We have computed a *compile-time function* to form a specific from a generic:

Carbon

```
fn MakeAverageSpecific(R:! Range where R.Value impls Numeric) -> <function> {
  let v0:! auto = R;
  let v1:! auto = v0.Value;
  let v2:! auto = (IntLiteral(0) as ImplicitAs(v1)).Convert;
  let v3:! auto = v0.Begin;
  let v4:! auto = v0.End;
  let v5:! auto = (v1 as AddAssign).Op;
  let v6:! auto = (v1 as DivWith(i32)).Op;
  return <make specific>(Average, (v0, v1, v2, v3, v4, v5, v6));
}
```

Forming a specific from a generic is compile-time function evaluation.

# Building a specific

```carbon
MakeAverageSpecific([i32; 3])
```

Carbon

```carbon
fn MakeAverageSpecific(R:! Range where R.Value impls Numeric) -> <function> {
  let v0:! auto = R;
  let v1:! auto = v1.Value;
  let v2:! auto = (IntLiteral(0) as ImplicitAs(v1)).Convert;
  let v3:! auto = v0.Begin;
  let v4:! auto = v0.End;
  let v5:! auto = (v1 as AddAssign).Op;
  let v6:! auto = (v1 as DivWith(i32)).Op;
  return <make specific>(Average, (v0, v1, v2, v3, v4, v5, v6));
}
```

Carbon

# Building a specific

```
MakeAverageSpecific([i32; 3])
```

Carbon

```
fn MakeAverageSpecific(R:! Range where R.Value impls Numeric) -> <function> {
  let v0:! auto = [i32; 3];
  let v1:! auto = i32;
  let v2:! auto = <builtin IntLiteral to i32 conversion>;
  let v3:! auto = [i32; 3].Begin;
  let v4:! auto = [i32; 3].End;
  let v5:! auto = <builtin AddAssign for i32>;
  let v6:! auto = <builtin DivWith for i32>;
  return <make specific>(Average, (v0, v1, v2, v3, v4, v5, v6));
}
```

Carbon

# Specific representation

## Generic

Carbon

```
Average[R:! Range where...]

inst[0] = R
inst[1] = #0.Value
inst[2] =
  (IntLiteral(0) as ImplicitAs(#1)).Convert
inst[3] = #0.Begin
inst[4] = #0.End
inst[5] = (#1 as AddAssign).Op
inst[6] = (#1 as DivWith(i32)).Op
```

## Specific

Carbon

```
Average with R = [i32; 3]

value[0] = [i32; 3];
value[1] = i32;
value[2] =
  <builtin IntLiteral to i32 conversion>;
value[3] = [i32; 3].Begin;
value[4] = [i32; 3].End;
value[5] = <builtin AddAssign for i32>;
value[6] = <builtin DivWith for i32>;
```

# Specific representation

## Generic

```
Average[R:! Range where...]

inst[0] = R
inst[1] = #0.Value
inst[2] =
  (IntLiteral(0) as ImplicitAs(#1)).Convert
inst[3] = #0.Begin
inst[4] = #0.End
inst[5] = (#1 as AddAssign).Op
inst[6] = (#1 as DivWith(i32)).Op
```

Carbon

## Specific

```
Average with R = [i32; 3]

value[0] = [i32; 3];
value[1] = i32;
value[2] =
  <builtin IntLiteral to i32 conversion>;
value[3] = [i32; 3].Begin;
value[4] = [i32; 3].End;
value[5] = <builtin AddAssign for i32>;
value[6] = <builtin DivWith for i32>;
```

Carbon

```
let a: [i32; 3] = (1, 2, 3);
let b: auto = Average(a);
```

Carbon

# Specific representation

## Generic

```
Average[R:! Range where...]                       Carbon

inst[0] = R
inst[1] = #0.Value
inst[2] =
  (IntLiteral(0) as ImplicitAs(#1)).Convert
inst[3] = #0.Begin
inst[4] = #0.End
inst[5] = (#1 as AddAssign).Op
inst[6] = (#1 as DivWith(i32)).Op
```

## Specific

```
Average with R = [i32; 3]                         Carbon

value[0] = [i32; 3];
value[1] = i32;
value[2] =
  <builtin IntLiteral to i32 conversion>;
value[3] = [i32; 3].Begin;
value[4] = [i32; 3].End;
value[5] = <builtin AddAssign for i32>;
value[6] = <builtin DivWith for i32>;
```

```
let a: [i32; 3] = (1, 2, 3);                      Carbon
let b: auto = Average(a);
```

- Look up return type of generic: `inst[1]`
- Look up `value[1]` in specific: `i32`

# Templates

So far, only talked about types and constant values that *symbolically* depend on generic parameters.

What about templates?

- Kind of instruction may depend on parameters
- Validity may depend on parameters too

# Templates

Add another kind of instruction to instantiate a single expression

```
fn CallF[template T:! type](x: T) {
  x.F();
}
```

Carbon

# Templates

Add another kind of instruction to instantiate a single expression

Carbon

```
fn CallF[template T:! type](x: T) {
  x.F();
}
```

# Templates

Add another kind of instruction to instantiate a single expression

Carbon

```
fn CallF[template T:! type](x: T) {
  #0();
}
```

- `#0` = `<instantiate member access>(`x`, `F`)`

# Templates

Add another kind of instruction to instantiate a single expression

Carbon

```carbon
fn CallF[template T:! type](x: T) {
  #0();
}
```

- `#0` = ``<instantiate member access>(`x`, `F`)``

# Templates

Add another kind of instruction to instantiate a single expression

Carbon

```
fn CallF[template T:! type](x: T) {
  #1;
}
```

- `#0` = `` <instantiate member access>(`x`, `F`) ``
- `#1` = `<instantiate call>(#0)`

# Templates

Add another kind of instruction to instantiate a single expression

```
fn CallF[template T:! type](x: T) {
  #1;
}
```

Carbon

- `#0` = `<instantiate member access>(`x`, `F`)`
- `#1` = `<instantiate call>(#0)`

Evaluating `<instantiate>` instruction produces another instruction

- Evaluation can fail

# Templates

Add another kind of instruction to instantiate a single expression

```carbon
fn CallF[template T:! type](x: T) {
  #1;
}
```

Carbon

- `#0` = `<instantiate member access>(`x`, `F`)`
- `#1` = `<instantiate call>(#0)`

Evaluating `<instantiate>` instruction produces another instruction

- Evaluation can fail

Not a dependent parse tree representing the eventual meaning of the program

- Instead, a *computation* that builds that meaning

# Templates

Forming a specific is still a compile-time function evaluation

- But have compile-time instruction that computes another instruction
- Useful metaprogramming tool in general

# Code complexity cost

Lose orthogonality

- Clang: `Expr*`, `Stmt*`, `Decl*`
  - Same for non-template and template
- Carbon: `pair<InstId, SpecificId>`
  - Must track `SpecificId` when navigating IR
  - Whole toolchain needs to know about generics

# Tradeoff

Clang dependent parse tree model:

- *Semantic representation* of templates
- *Orthogonality*

Carbon toolchain overlay model:

- *Smaller representation*
- *Faster instantiation*

# Tradeoff

Clang dependent parse tree model:

- *Semantic representation* of templates
- *Orthogonality*

Carbon toolchain overlay model:

- *Smaller representation*: 1.2KiB -> 120B (~10x)
- *Faster instantiation*

# Tradeoff

Clang dependent parse tree model:

- *Semantic representation* of templates
- *Orthogonality*

Carbon toolchain overlay model:

- *Smaller representation*: 1.2KiB -> 120B (~10x)
- *Faster instantiation*: 43μs -> 4μs (~10x)

# Tradeoff

Clang dependent parse tree model:

- *Semantic representation* of templates
- *Orthogonality*

Carbon toolchain overlay model:

- *Smaller representation*: 1.2KiB -> 120B (~10x)
- *Faster instantiation*: 43µs -> 4µs (~10x)
- Supports *lowering optimizations* (not implemented yet)

https://docs.carbon-lang.dev/#join-us

# Questions?

Slides: https://chandlerc.blog/slides/2024-llvm-generic-implementation

Image credit: https://unsplash.com/photos/a-blue-background-with-lines-and-dots-xuTJZ7uD7PI

# Bonus slides: token soup

# Token soup

Parse: collect list of tokens

```cpp
template<typename T> T clamp(const T &a)
  = { "return" "a" "<" "T" "(" ")" "?" "T" "(" ")" ":" "a" ";" }
```

C++

# Token soup

Parse: collect list of tokens

```cpp
template<typename T> T clamp(const T &a)
  = { "return" "a" "<" "T" "(" ")" "?" "T" "(" ")" ":" "a" ";" }
```

Instantiate: replay tokens

```cpp
template<> int clamp<int>(const int &a) {
  return a < int() ? int() : a;
}
```

Or:

```cpp
using T = int;
template<> T clamp<T>(const T &a) {
  return a < T() ? T() : a;
}
```

# Token soup

Good:

- Simple: reuses components you already had
- Orthogonal: rest of frontend doesn’t need to know
- “Parsing” templates is very cheap
- *Permissive* and *compatible*: can choose how to interpret code late
  - No need for `typename X::template Y<...>`
  - Better *error recovery*

# Token soup

Good:

- Simple: reuses components you already had
- Orthogonal: rest of frontend doesn’t need to know
- “Parsing” templates is very cheap
- *Permissive* and *compatible*: can choose how to interpret code late
    - No need for `typename X::template Y<...>`
    - Better *error recovery*

Bad:

- Incomplete (example: redeclaration matching)
- Pay full cost for each instantiation

# Token soup

Good:

- Simple: reuses components you already had
- Orthogonal: rest of frontend doesn’t need to know
- “Parsing” templates is very cheap
- *Permissive* and *compatible*: can choose how to interpret code late
  - No need for `typename X::template Y<...>`
  - Better *error recovery*

Bad:

- Incomplete (example: redeclaration matching)
- Pay full cost for each instantiation
- Wrong

# Token soup

C++

```cpp
int a = 1;
namespace N {
  template<typename T> int f() { return a; }
  int a = 2;
}
int b = N::f<int>();
```

# Token soup

```cpp
int a = 1;
namespace N {
  template<typename T> int f() { return a; }
  int a = 2;
}
int b = N::f<int>();
```

EDG:

- Name lookup during instantiation ignores things declared later
- Prototype instantiation immediately after definition
  - Diagnose templates with syntax errors
  - Collect information from template definition context and annotate on tokens

# Token soup

C++

```cpp
int a = 1;
namespace N {
  template<typename T> int f() { return a; }
  int a = 2;
}
int b = N::f<int>();
```

EDG:

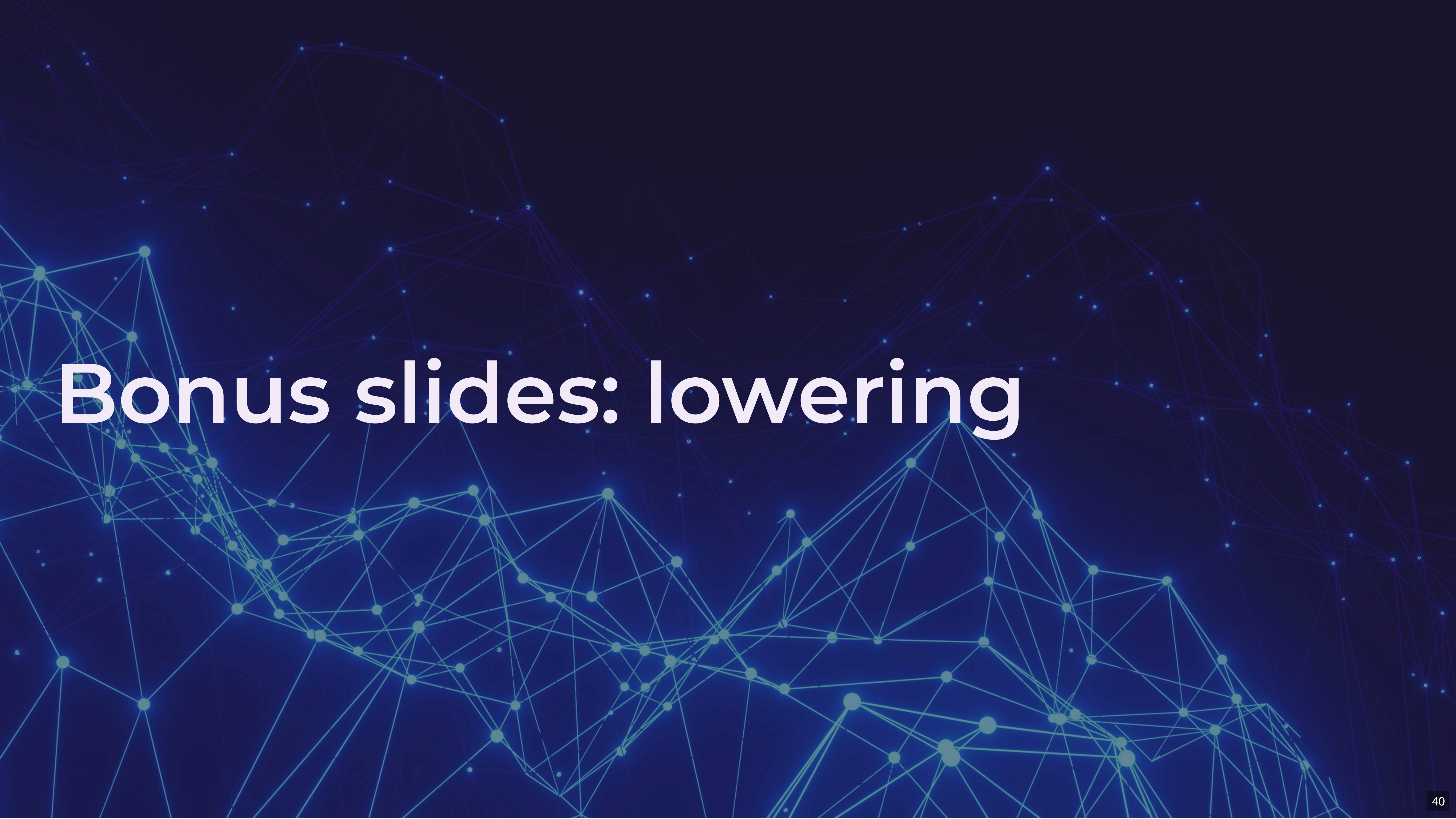
- Name lookup during instantiation ignores things declared later
- Prototype instantiation immediately after definition
  - Diagnose templates with syntax errors
  - Collect information from template definition context and annotate on tokens

MSVC (old parser):

- `b == 2`

## Token soup

- Easy to implement
- Hard to implement well



# Bonus slides: lowering

# Lowering

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #1)
    -> #2 {
  var n: i32 = 0;
  var sum: #2 = 0.(#3)();
  for (elem in v.(#4)() ... v.(#5)()) {
    sum.(#6)(elem);
    ++n;
  }
  return sum.(#7)(if n != 0 then n else 1);
}
```

# Lowering

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #1)
    -> #2 {
  var n: i32 = 0;
  var sum: #2 = 0.(#3)();
  for (elem in v.(#4)() ... v.(#5)()) {
    sum.(#6)(elem);
    ++n;
  }
  return sum.(#7)(if n != 0 then n else 1);
}
```

->

```
define void @_CAverage.Main.abc123() {
entry:
  %n.var = alloca i32, align 4
  store i32 0, ptr %n.var, align 4
  %sum.var = alloca
```

# Lowering

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #1)
    -> #2 {
  var n: i32 = 0;
  var sum: #2 = 0.(#3)();
  for (elem in v.(#4)() ... v.(#5)()) {
    sum.(#6)(elem);
    ++n;
  }
  return sum.(#7)(if n != 0 then n else 1);
}
```

->

```
define void @_CAverage.Main.abc123() {
entry:
  %n.var = alloca i32, align 4
  store i32 0, ptr %n.var, align 4
  %sum.var = alloca
```

- `#2` = `i32` (Carbon)

# Lowering

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #1)
    -> #2 {
  var n: i32 = 0;
  var sum: #2 = 0.(#3)();
  for (elem in v.(#4)() ... v.(#5)()) {
    sum.(#6)(elem);
    ++n;
  }
  return sum.(#7)(if n != 0 then n else 1);
}
```

->

```
define void @_CAverage.Main.abc123() {
entry:
  %n.var = alloca i32, align 4
  store i32 0, ptr %n.var, align 4
  %sum.var = alloca
```

- `#2` = `i32` (Carbon)
- Lowers to `i32` (LLVM)

# Lowering

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #1)
    -> #2 {
  var n: i32 = 0;
  var sum: #2 = 0.(#3)();
  for (elem in v.(#4)() ... v.(#5)()) {
    sum.(#6)(elem);
    ++n;
  }
  return sum.(#7)(if n != 0 then n else 1);
}
```

# Lowering

Carbon

```
fn Average[R:! Range where R.Value impls Numeric](v: #1)
    -> #2 {
  var n: i32 = 0;
  var sum: #2 = 0.(#3)();
  for (elem in v.(#4)() ... v.(#5)()) {
    sum.(#6)(elem);
    ++n;
  }
  return sum.(#7)(if n != 0 then n else 1);
}
```

->

```
define void @_CAverage.Main.abc123(%v.param: ptr) {
entry:
  %n.var = alloca i32, align 4
  store i32 0, ptr %n.var, align 4
  %sum.var = alloca i32, align 4
  store i32 0, ptr %sum.var, align 4
  ...
```

# Lowering

- Track which slots are lowered, and the lowered values

```
#1 -> ptr
#2 -> i32
#3 -> <builtin implicit conversion from IntLiteral to i32>
#4 -> @_CBegin.Array.Core.abc123
#5 -> @_CEnd.Array.Core.abc123
#6 -> <builtin AddAssign for i32>
#7 -> <builtin DivWith for i32>
```

# Lowering

- Track which slots are lowered, and the lowered values

```
#1 -> ptr
#2 -> i32
#3 -> <builtin implicit conversion from IntLiteral to i32>
#4 -> @_CBegin.Array.Core.abc123
#5 -> @_CEnd.Array.Core.abc123
#6 -> <builtin AddAssign for i32>
#7 -> <builtin DivWith for i32>
```

- When lowering the same generic again, check for matching lowered values and reuse

# Lowering

- Track which slots are lowered, and the lowered values

```
#1 -> ptr
#2 -> i32
#3 -> <builtin implicit conversion from IntLiteral to i32>
#4 -> @_CBegin.Array.Core.abc123
#5 -> @_CEnd.Array.Core.abc123
#6 -> <builtin AddAssign for i32>
#7 -> <builtin DivWith for i32>
```

- When lowering the same generic again, check for matching lowered values and reuse
- Use a fingerprint of the lowered values in the decorated name of the specific

# Lowering

Result:

- Specifics with the same generic and same overlays lowered to the same function
- Example: `Vector(i32*).Size` and `Vector(String*).Size` are the same function

# Lowering

Result:

- Specifics with the same generic and same overlays lowered to the same function
- Example: `Vector(i32*).Size` and `Vector(String*).Size` are the same function

Overlay model gives us the information to do this

- List of things that vary between specifics
- Per-specific lowered value