Weak Hiding for C++ Concepts
and a Generic Way to Understand Name Binding

Larisse Voufo, Marcin Zalewski, Jeremiah Willcock and Andrew Lumsdaine
Center for Research in Extreme Scale Technologies (CREST)
Indiana University

C++Now 2013, Aspen, CO, USA

Objectives

❖ A simpler way to describe name binding.

❖ A more powerful way to describe name binding.

❖ Applying this description to C++ concepts.

Our Tools

- scope combinators,

- an abstraction of programming languages, and

- a distinction between **name lookup** and **name resolution**.

Outline

① Current name binding mechanisms:

1. Across languages.
2. Specific to C++:
   a. Argument-dependent lookup (ADL).
   b. Uses of operators.

② A new scoping rule: **Weak hiding**

- Implementation: **two-stage name binding (Bind^{x2})**
- Application: Preventing code breakage when switching to concepts-enabled libraries.

Name Binding

- Deciding what a particular use of a name refers to.
- Use of a name = **a *reference***; What it refers to = **a *declaration***.

```
void foo();
void foo(int, double);



void test() {

  foo();
}
```

Name Binding

❖ Binds **a** ***reference*** to **a** ***declaration***.

```
void foo();
void foo(int, double);



namespace ns { void foo(int); }

void test() {
    using ns::foo;
    foo();
}
```

Name Binding

- Binds **a *reference*** to **a *declaration***.

```
void foo();

namespace adl { struct X {};
                void foo(X); }

namespace ns { void foo(int); }

void test(adl::X x) {
   using ns::foo;
   foo(x);
}
```

Name Binding

❖ Binds **a *reference*** to **a *declaration***.

```
concept Foo2<typename P> =
   requires (P a, P b) { foo(a, b); }

void foo(int) { }


template<Foo2 T>
void gen_func(T a, T b) {
   foo(a, b);
   foo(1);
}
```

Name Binding

- Depends on **a scope**, nested or combined with other scopes.

```
void foo();

namespace adl { struct X {};
                void foo(X); }

namespace ns { void foo(int); }

void test(adl::X x) {
    using ns::foo;
    foo(x);
}
```

::

adl

ns

test

Name Binding

❖ Differs between **languages**, **designs**, or **kinds of references**:

- argument-dependent lookup (ADL),
- simple function calls,
- uses of operators,
- uses of types,
- C++ Multimethods,
- C++,
- Haskell,
- etc…

```
void foo();

namespace adl { struct X {};
                void foo(X); }

namespace ns { void foo(int); }

void test(adl::X x, adl::X y) {
    using ns::foo;
    foo(x + y);
}
```

::
adl
ns
test

Standards for Name Binding

- Can be challenging to understand, analyze, and compare.
- Are intertwined with other sections of the standard.

- **C++ Name Binding Specification, at a glance**:

3.4 – Name lookup	13 pages, 45+ paragraphs
14.6 – Name resolution	13 pages, 50+ paragraphs
13 – Overloading	30 pages, 140+ paragraphs
11 – Member access	10 pages, 35+ paragraphs

Standards for Name Binding

❖ Typically depend on details of languages and the varied features they support.

ConceptC++:

```
template<Foo2 T>
void test(T) {
   foo();
}
```

⇒ C++ templates + Concepts

⇒ (Non)dependent names

⇒ Overload resolution

⇒ Namespaces + `using` declarations

⇒ C++ compilers, e.g., GCC, Clang, …

⇒ Etc…

Haskell:

```
test::Foo2 t => t -> ()
test _ = foo()
```

⇒ Hindley-Milner type system + Type classes

⇒ Type inference + Dependency analysis

⇒ Dictionary-passing style

⇒ Modules and `import` declarations

⇒ Haskell compilers, e.g., GHC, Hugs, NHC, …

⇒ Etc…

Standards for Name Binding

❖ Are understood differently by different compilers.

```
namespace adl { struct X {};
                void foo(X); }

namespace ns { void foo(int); }

void bar(adl::X x) {
  using ns::foo;
  foo(x);
}

void baz(adl::X x) {
  void foo(int);
  foo(x);
}
```

In bar():

Success. ADL enabled.
Binds `foo(x)` to `adl::foo()`.

In baz():

GCC, Clang, Comeau:

Failure. ADL Disabled.
`foo(x)` does not match `baz::foo(int)`.

Intel:

Success.

Standards for Name Binding

- Are understood differently by different compilers.

```
struct X {}; struct Y {};

void operator+(X, X) { }
void operator+(X, Y) { }

void test(X x, Y y) {
  void operator+(X, X);
  x + x;
  x + y;
  operator+(x, x);
  operator+(x, y);
}
```

GCC, Clang, MS Visual Studio editor :

`x + y` succeeds.
`operator+(x, y)` fails.

Intel, MS Visual Studio:

`x + y` succeeds.
`operator+(x, y)` succeeds.

Comeau:

`x + y` fails.
`operator+(x, y)` fails.

Standards for Name Binding

- Gain complexity with the addition of concepts,
- breaking seemingly valid codes.

```
concept Foo2<typename T> =
   requires (T a, T b) { foo(a, b); }

void  foo(int) { }

template<Foo2 T>
void gen_func(T a, T b) {
   foo(a, b);
   foo(1);
}
```

Name Binding Specifications

- Gain complexity with the addition of concepts,
- breaking seemingly valid codes.

```
concept Foo2<typename P> =
   requires (P a, P b) { foo(a, b); }

void foo(int) { }

template<Foo2 T>
void gen_func(T a, T b) {
   foo(a, b);
   foo(1);
}
```

::

Foo2<P>

Foo2<T>

template…

gen_func<T>

Name Binding Specifications

- Gain complexity with the addition of concepts,
- breaking seemingly valid codes.

```
concept Foo2<typename P> =
   requires (P a, P b) { foo(a, b); }

void foo(int) { }

template<Foo2 T>
void gen_func(T a, T b) {
   foo(a, b);
   foo(1);
}
```

::

Foo2<T>

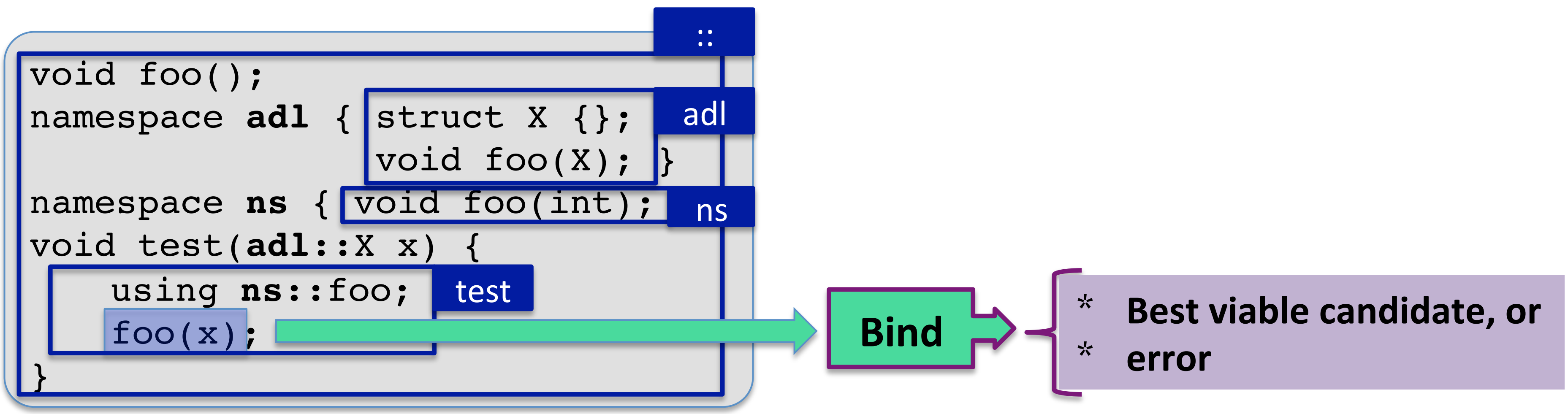
Should `Foo2<T>::foo()` shadow `::foo()`? How?

Reject or accept the call `foo(1)`?

CENTER FOR RESEARCH IN EXTREME SCALE TECHNOLOGIES
INDIANA UNIVERSITY
Pervasive Technology Institute

Our Name Binding Framework

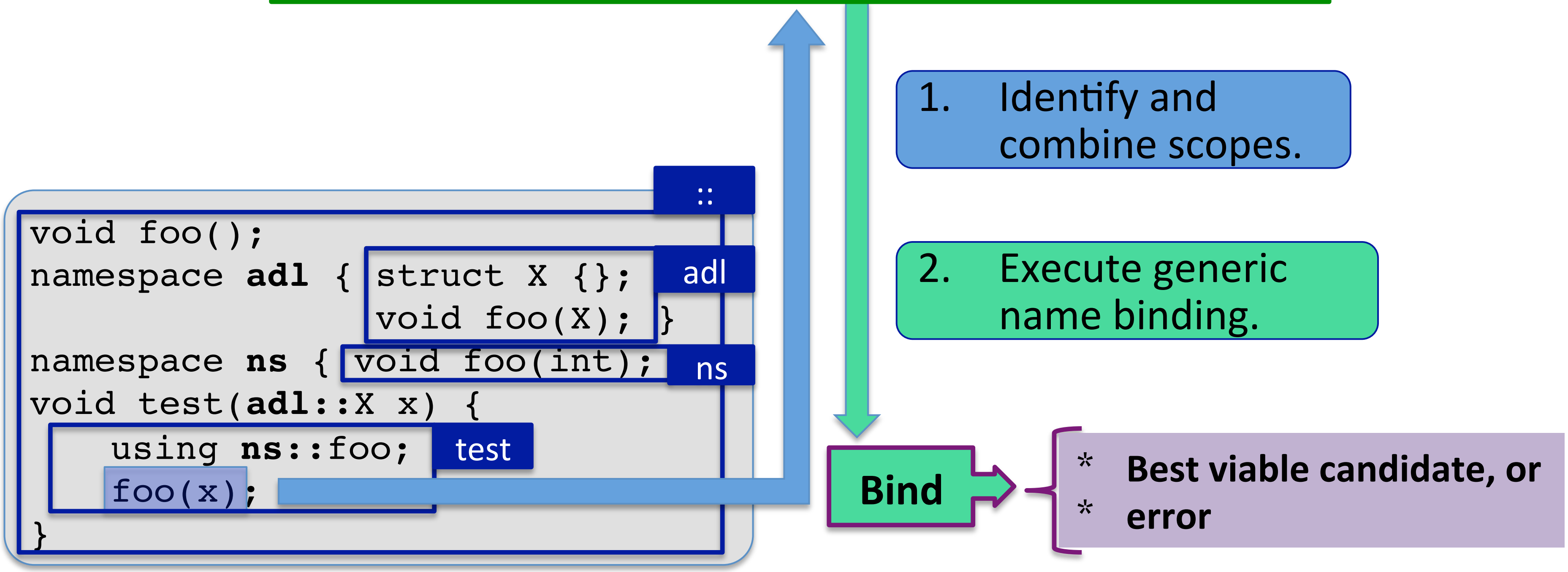
❖ Specifies name binding.



Our Name Binding Framework

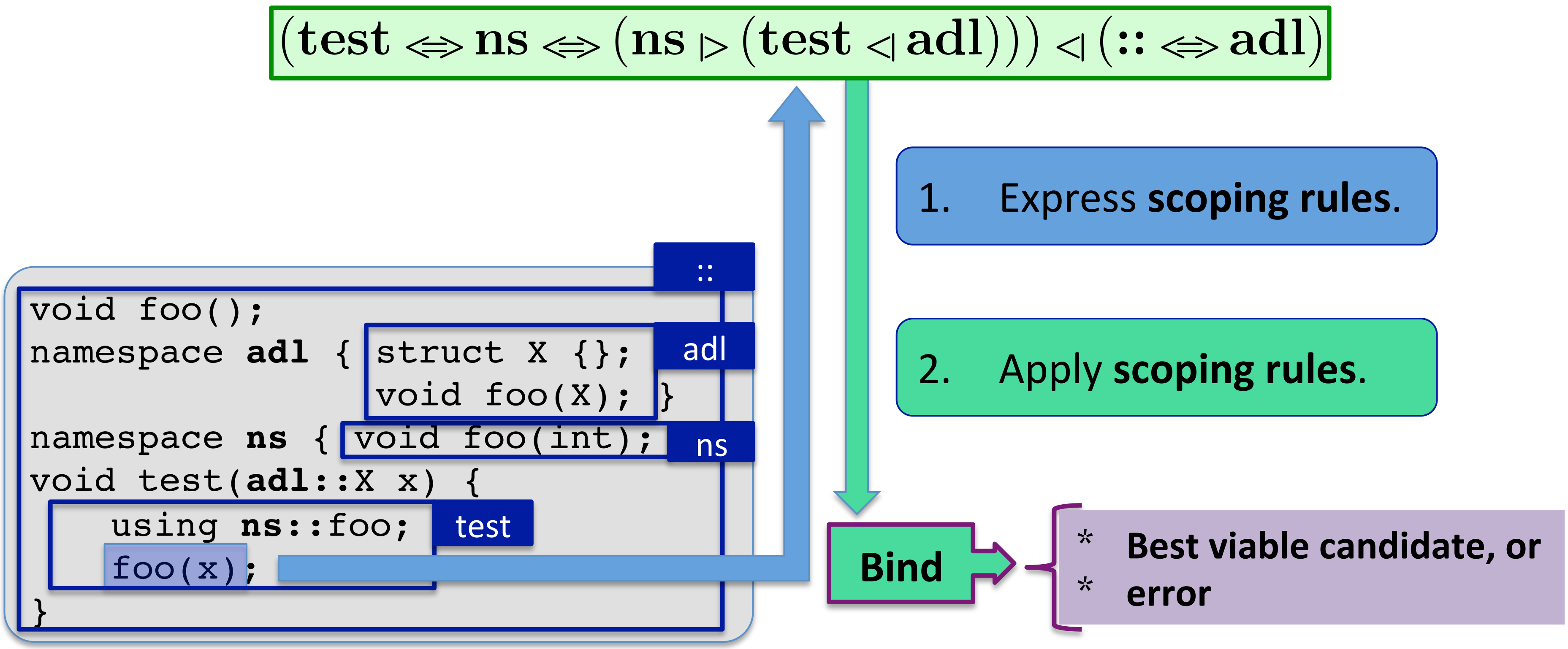
❖ Specifies name binding, independently of the language.

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \rhd (\mathbf{test} \lhd \mathbf{adl}))) \lhd (\mathbf{::} \Leftrightarrow \mathbf{adl})$$



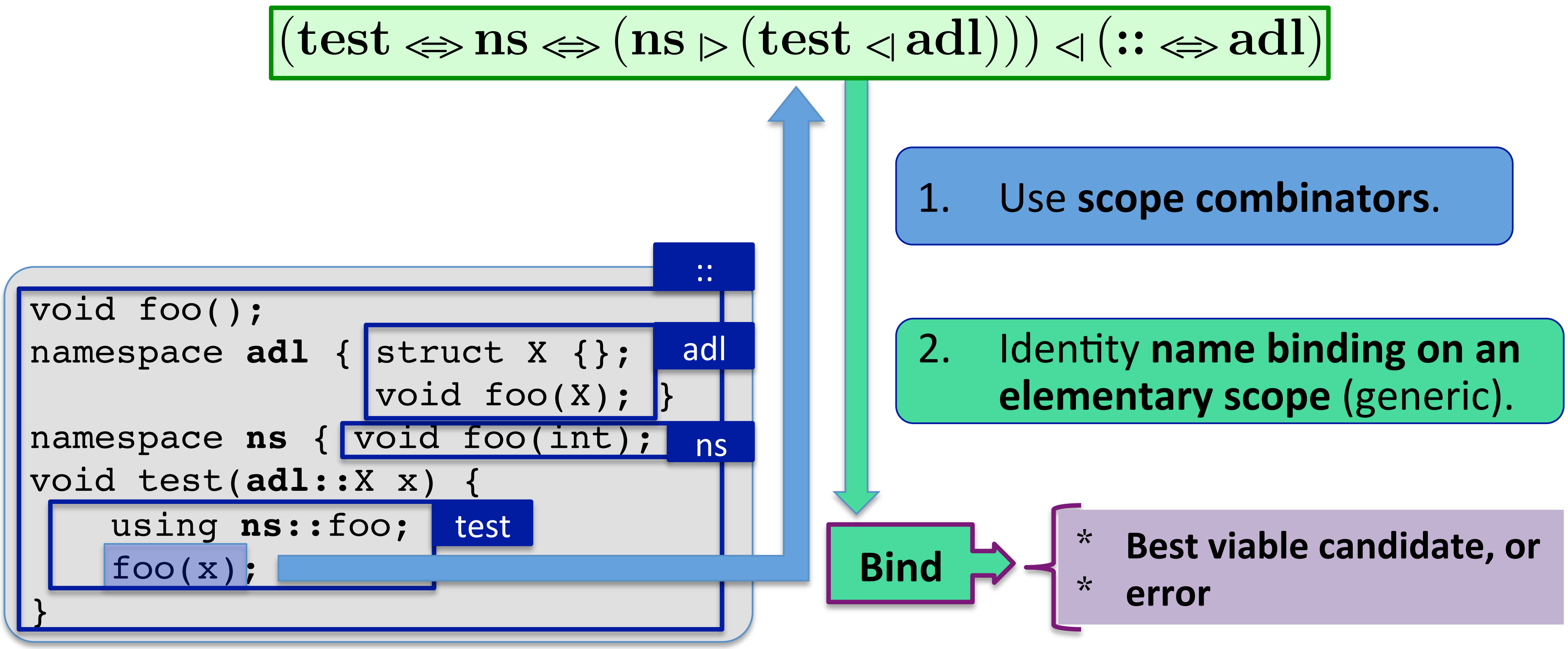
Our Name Binding Framework

❖ Specifies name binding, independently of the language.



Our Name Binding Framework

❖ Specifies name binding, independently of the language.



Our Name Binding Framework

❖ Specifies name binding, independently of the language.

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \triangleright (\mathbf{test} \triangleleft \mathbf{adl}))) \triangleleft (\mathbf{::} \Leftrightarrow \mathbf{adl})$$

```
void foo();
namespace adl { struct X {};
                void foo(X); }
namespace ns { void foo(int); }
void test(adl::X x) {
    using ns::foo;
    foo(x);
}
```

::

adl

ns

test

1. Use **scope combinators**.
2. Depends on a **Language** concept.

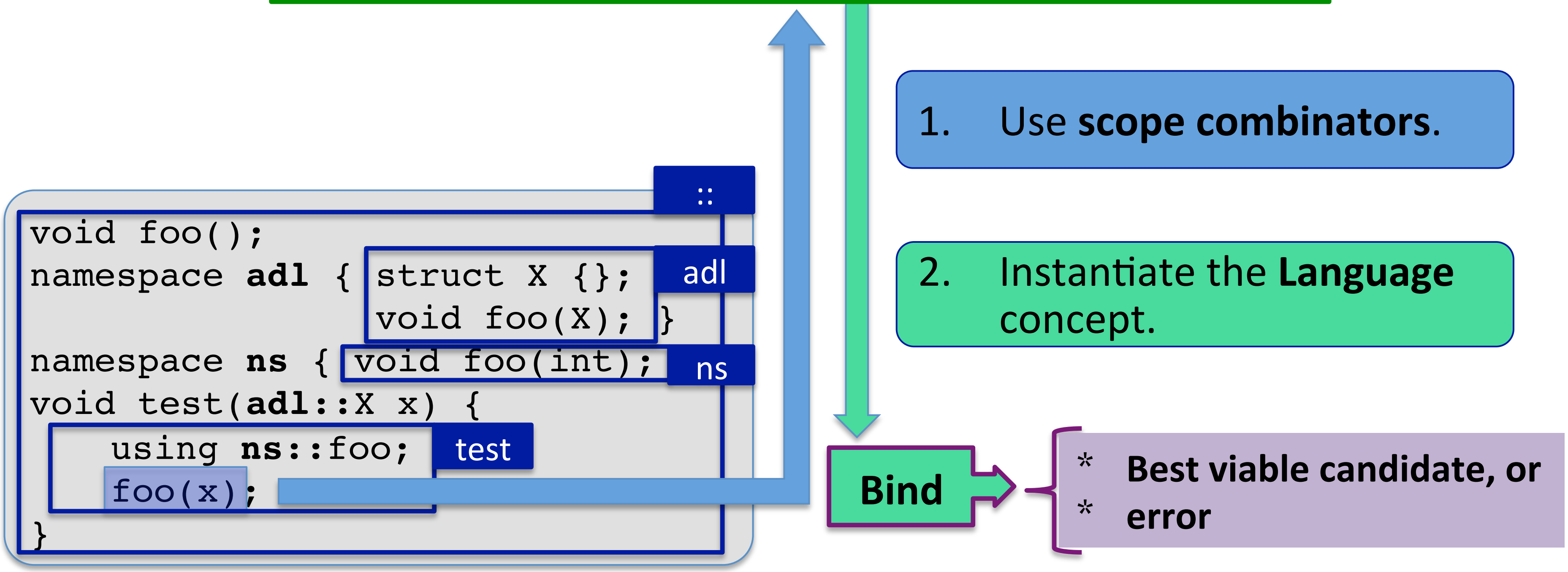
Bind

* **Best viable candidate, or**
* **error**

Our Name Binding Framework

❖ Specifies name binding, independently of the language.

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \triangleright (\mathbf{test} \triangleleft \mathbf{adl}))) \triangleleft (\mathbf{::} \Leftrightarrow \mathbf{adl})$$



Our Name Binding Framework

❖ Abstracts from *declarations*, *references*, and *scopes*.

$$\boxed{\mathbf{bind} : \mathbf{Ref} \times \mathbf{Scope}_{\mathbf{Ref},\mathbf{Decl}} \rightarrow (\mathbf{Decl} + \mathrm{Error})}$$

```
void foo();
namespace adl { struct X {};
                void foo(X); }
namespace ns { void foo(int); }
void test(adl::X x) {
    using ns::foo;
    foo(x);
}
```

Scopes as maps of **references** to **sets of matching declarations**.

Bind

* **Best viable candidate, or**
* **error**

Our Name Binding Framework

- Views name binding as composed of **name lookup** and **resolution**.

$$\textbf{bind} : \textbf{Ref} \times \textbf{Scope}_{\textbf{Ref},\textbf{Decl}} \rightarrow (\textbf{Decl} + \text{Error})$$

$$\boldsymbol{bind}\ (ref, scope) = ((\boldsymbol{resolve}\ ref) \circ (\boldsymbol{lookup}\ ref))\ scope$$

```
void foo();
namespace adl { struct X {};
                void foo(X); }
namespace ns { void foo(int); }
void test(adl::X x) {
    using ns::foo;
    foo(x);
}
```

Name lookup returns the **set of matching declarations**, for a given **reference**.

Bind

- Best viable candidate, or
- error

1. Expressing Scoping Rules

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \triangleright (\mathbf{test} \triangleleft \mathbf{adl}))) \triangleleft (\mathbf{::} \Leftrightarrow \mathbf{adl})$$

```
void foo();
namespace adl { struct X {};
                void foo(X); }
namespace ns { void foo(int); }
void test(adl::X x) {
    using ns::foo;
    foo(x);
}
```

::

adl

ns

test

1. Use **scope combinators**.

2. Instantiate the **Language** concept.

Bind

- Best viable candidate, or
- error

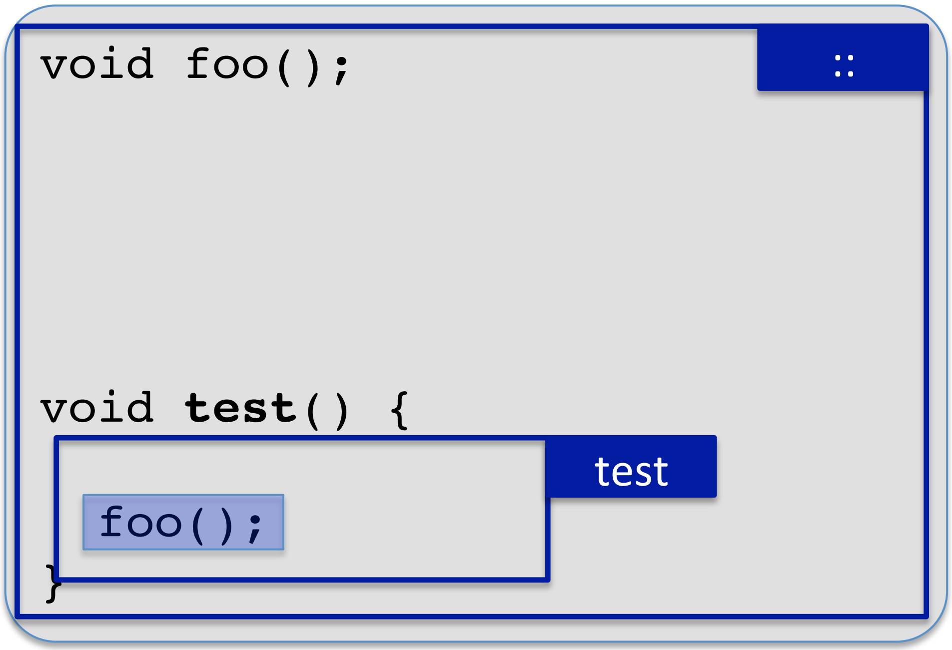
$$\mathrm{Scope}_{\mathrm{Ref,Decl}} \times \mathrm{Scope}_{\mathrm{Ref,Decl}} \rightarrow \mathrm{Scope}_{\mathrm{Ref,Decl}}$$

The Combinators

- ***Hiding***: $\lhd$
  - Commonly known as "shadowing".
- ***Merging***: $\Longleftrightarrow$
  - Usually the alternative option to "shadowing".
- ***Opening***: $\rhd$
  - [New name] Necessary to describe ADL.
  - A dual of *hiding*.
- ***Weak Hiding***: $\Leftarrow\!|$
  - [New rule] Necessary for (C++) concepts.
  - A sweet middle between *hiding* and merging.

The *Hiding* Combinator ($\lhd$)

$$\mathbf{s_1} \lhd \mathbf{s_2} = lookup_{ref}\ \mathbf{s_1}\ ?\ lookup_{ref}\ \mathbf{s_1} : lookup_{ref}\ \mathbf{s_2}$$



$$\text{test} \lhd \text{::}$$

<u>Result:</u>

Binds **`foo()`** to **`::foo()`**.

The *Merging* Combinator ($\Leftarrow\!\!\!\Rightarrow$)

$$\mathbf{s_1} \Leftarrow\!\!\Rightarrow \mathbf{s_2} = \boldsymbol{lookup}_{ref}\, \mathbf{s_1} \cup \boldsymbol{lookup}_{ref}\, \mathbf{s_2}$$

```
void foo();

namespace ns {
  void foo(int);
}

void test() {
  using ns::foo;
  foo();
}
```

::

ns

test

$$(\mathbf{test} \Leftarrow\!\!\Rightarrow \mathbf{ns}) \lhd \mathbf{::}$$

<u>Result:</u>

Finds **`ns::foo()`**;

Fails to bind **`foo()`**.

The *Weak Hiding* Combinator ($\Leftarrow\!\!|$)

$$\mathbf{s_1} \Leftarrow\!\!| \ \mathbf{s_2} = \left(\mathit{resolve}_{ref} \circ \mathit{lookup}_{ref}\right) \mathbf{s_1} \ ? \ \mathit{lookup}_{ref}\, \mathbf{s_1} \ : \ \mathit{lookup}_{ref}\, \mathbf{s_2}$$

```
void foo();                          ::

namespace ns {
  void foo(int);          ns
}

void test() {
  using ns::foo;          test
  foo();
}
```

$$(\mathbf{test} \Leftrightarrow \mathbf{ns}) \Leftarrow\!\!| \ \mathbf{::}$$

Result:

Binds **`foo()`** to **`::foo()`**.

The *Opening* Combinator ($\rhd$)

$$\mathbf{s}_1 \rhd \mathbf{s}_2 = \mathit{lookup}_{ref}\ \mathbf{s}_1\ ?\ \mathit{lookup}_{ref}\ \mathbf{s}_2 : \mathbf{empty}$$

```
void foo();

namespace ns { void foo(int); }

namespace adl { struct X {};
                void foo(typ); }

void test(adl::X x) {
  using ns::foo;

  foo(x);
}
```

:: ns adl test

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \rhd (\mathbf{test} \lhd \mathbf{adl}))) \lhd (:: \Leftrightarrow \mathbf{adl})$$

Result:

Finds **`ns::foo()`**;

Enables ADL;

Binds **`foo(x)`** to **`adl::foo()`**.

The *Opening* Combinator ($\triangleright$)

$$\mathrm{s}_1 \triangleright \mathrm{s}_2 = \mathit{lookup}_{ref}\ \mathrm{s}_1 \ ?\ \mathit{lookup}_{ref}\ \mathrm{s}_2 : \mathbf{empty}$$

```
void foo();

namespace ns { void foo(int); }

namespace adl { struct X {};
                void foo(typ); }

void test(adl::X x) {
  using ns::foo;
  void foo();
  foo(x);
}
```

::

ns

adl

test

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \triangleright (\mathbf{test} \triangleleft \mathbf{adl}))) \triangleleft (:: \Leftrightarrow \mathbf{adl})$$

Result:

Finds **`test::foo()`**;

Disables ADL;

Fails to bind **`foo(x)`**.

The Combinators – Recap

$\boxed{\lhd\!\mid\ ,\ \Leftarrow\!\!\!\!\Rightarrow\ ,\ \mid\!\rhd\ ,\ \Leftarrow\!\mid}\ \boxed{\mathbf{Scope}_{\text{Ref,Decl}} \times \mathbf{Scope}_{\text{Ref,Decl}} \rightarrow \mathbf{Scope}_{\text{Ref,Decl}}}$

$$\mathbf{s_1} \Leftarrow\!\!\!\!\Rightarrow \mathbf{s_2} = \mathit{lookup}_{ref}\ \mathbf{s_1} \cup \mathit{lookup}_{ref}\ \mathbf{s_2}$$

$$\mathbf{s_1} \lhd \mathbf{s_2} = \mathit{lookup}_{ref}\ \mathbf{s_1}\ ?\ \mathit{lookup}_{ref}\ \mathbf{s_1}\ :\ \mathit{lookup}_{ref}\ \mathbf{s_2}$$

$$\mathbf{s_1} \rhd \mathbf{s_2} = \mathit{lookup}_{ref}\ \mathbf{s_1}\ ?\ \mathit{lookup}_{ref}\ \mathbf{s_2}\ :\ \mathbf{empty}$$

$$\mathbf{s_1} \Leftarrow\!\mid \mathbf{s_2} = \left(\mathit{resolve}_{ref} \circ \mathit{lookup}_{ref}\right) \mathbf{s_1}\ ?\ \mathit{lookup}_{ref}\ \mathbf{s_1}\ :\ \mathit{lookup}_{ref}\ \mathbf{s_2}$$

Special cases of a "*conditional*" combinator.

Applications

- Understanding current name binding mechanisms:
  - Argument-dependent lookup (ADL).
  - C++ operators.
  - A cross-language analysis.
- Exploring concepts designs
  - Understanding the current limitations.
  - Exploring new solutions:
    - weak hiding,
    - 2-Stage name binding, and
    - parameterized weak hiding.
- Simplifying compiler designs.

ADL Example

```
namespace ns1 { struct X {};
                void foo(X); }

namespace ns2 { void foo(int); }

void bar(ns1::X x) {
  using ns2::foo;
  foo(x);
}

void baz(ns1::X x) {
  void foo(int);
  foo(x);
  {                          // H
    using ns2::foo;
    foo(x);
  }
}
```

ns1 ns2 bar baz H ::

In bar():

In baz():

In baz()'s inner scope, H:

ADL Example

```
namespace ns1 { struct X {};
                void foo(X); }

namespace ns2 { void foo(int); }

void bar(ns1::X x) {
  using ns2::foo;
  foo(x);
}

void baz(ns1::X x) {
  void foo(int);
  foo(x);
  {                         // H
    using ns2::foo;
    foo(x);
  }
}
```

ns1

ns2

bar

::

In bar():

$$\boxed{(\mathbf{bar} \Leftrightarrow \mathbf{ns2} \Leftrightarrow (\mathbf{ns2} \triangleright (\mathbf{bar} \triangleleft \mathbf{ns1}))) \triangleleft (:: \Leftrightarrow \mathbf{ns1})}$$

In baz():

In baz()'s inner scope, H:

ADL Example

```
namespace ns1 { struct X {};
                void foo(X); }

namespace ns2 { void foo(int); }

void bar(ns1::X x) {
  using ns2::foo;
  foo(x);
}

void baz(ns1::X x) {
  void foo(int);
  foo(x);
  {
    using ns2::foo;
    foo(x);                    // H
  }
}
```

<u>In bar():</u>

$$\boxed{\begin{gathered}(\mathbf{bar} \Leftrightarrow \mathbf{ns2} \Leftrightarrow (\mathbf{ns2} \rhd (\mathbf{bar} \lhd \mathbf{ns1})))\\ \lhd (\mathbf{::} \Leftrightarrow \mathbf{ns1})\end{gathered}}$$

<u>In baz():</u>

$$\boxed{\mathbf{baz} \lhd (\mathbf{::} \Leftrightarrow \mathbf{ns1})}$$

<u>In baz()'s inner scope, H:</u>

ADL Example

```
namespace ns1 { struct X {};
                void foo(X); }

namespace ns2 { void foo(int); }

void bar(ns1::X x) {
  using ns2::foo;
  foo(x);
}

void baz(ns1::X x) {
  void foo(int);
  foo(x);
  {
    using ns2::foo;
    foo(x);
  }                                  // H
}
```

ns1 ns2 baz H ::

In bar():

$$\boxed{\begin{gathered}(\mathbf{bar} \Leftrightarrow \mathbf{ns2} \Leftrightarrow (\mathbf{ns2} \rhd (\mathbf{bar} \lhd \mathbf{ns1})))\\ \lhd (\mathbf{::} \Leftrightarrow \mathbf{ns1})\end{gathered}}$$

In baz():

$$\boxed{\mathbf{baz} \lhd (\mathbf{::} \Leftrightarrow \mathbf{ns1})}$$

In baz()'s inner scope, H:

$$\boxed{\begin{gathered}(\mathbf{H} \Leftrightarrow \mathbf{ns2} \Leftrightarrow (\mathbf{ns2} \rhd (\mathbf{H} \lhd \mathbf{ns1})))\\ \lhd \mathbf{baz} \lhd (\mathbf{::} \Leftrightarrow \mathbf{ns1})\end{gathered}}$$

ADL Example

```
namespace ns1 { struct X {};
                void foo(X); }

namespace ns2 { void foo(int); }

void bar(ns1::X x) {
  using ns2::foo;
  foo(x);
}

void baz(ns1::X x) {
  void foo(int);
  foo(x);
  {                       // H
    using ns2::foo;
    foo(x);
  }
}
```

ns1

ns2

bar

baz

H

::

In bar():

$$\boxed{\mathbf{f}(\mathbf{bar}) \lhd (\mathbf{::} \Leftrightarrow \mathbf{ns1})}$$

In baz():

$$\boxed{\mathbf{f}(\mathbf{baz}) \lhd (\mathbf{::} \Leftrightarrow \mathbf{ns1})}$$

In baz()'s inner scope, H:

$$\boxed{\mathbf{f}(\mathbf{H}) \lhd \mathbf{f}(\mathbf{baz}) \lhd (\mathbf{::} \Leftrightarrow \mathbf{ns1})}$$

------------------ with ------------------

$$\begin{aligned} \mathbf{f}(\mathbf{X}) &= \mathbf{X} \Leftrightarrow \mathbf{U_X} \Leftrightarrow (\mathbf{U_X} \rhd (\mathbf{X} \lhd \mathbf{ns1})) \\ \mathbf{U_{bar}} = \mathbf{U_H} &= \mathbf{ns2} \\ \mathbf{U_{baz}} &= \mathbf{empty} \end{aligned}$$

ADL Scoping Rules

$$\mathbf{f_{ADL}}\left(\mathbf{H}\right) \lhd \Lbag_{i=1}^{s} \mathbf{f_{ADL}}\left(\mathbf{S_i}\right) \lhd \Lbag_{i=1}^{n-1} \mathbf{fn_{ADL}}\left(\mathbf{N_i}\right) \lhd \left(\mathbf{N_n} \Leftrightarrow \mathbf{U_{N_n}} \Leftrightarrow \left(\left(\widetilde{\mathbf{N}}_\mathbf{n} \Leftrightarrow \widetilde{\mathbf{U}}_{\mathbf{N_n}}\right) \lhd \mathbf{ADL}\right)\right)$$

$$\mathbf{f_{ADL}}\left(\mathbf{X}\right) = \mathbf{X} \Leftrightarrow \mathbf{U_X} \Leftrightarrow \left(\mathbf{U_X} \rhd \left(\left(\mathbf{X} \Leftrightarrow \widetilde{\mathbf{U}}_\mathbf{X}\right) \lhd \mathbf{ADL}\right)\right),$$

$$\mathbf{S_1} \cdots \mathbf{S_s} = \text{surrounding non-namespace scopes},$$

$$\mathbf{fn_{ADL}}\left(\mathbf{N}\right) = \mathbf{N} \Leftrightarrow \mathbf{U_N} \Leftrightarrow \left(\left(\mathbf{N} \Leftrightarrow \mathbf{U_N}\right) \rhd \left(\left(\widetilde{\mathbf{N}} \Leftrightarrow \widetilde{\mathbf{U}}_\mathbf{N}\right) \lhd \mathbf{ADL}\right)\right),$$

$$\mathbf{N_1} \cdots \mathbf{N_n} = \text{surrounding namespaces},$$

$$\mathbf{H} = \text{scope where name binding is triggered from},$$

$$\mathbf{U_X} = \text{using declarations in scope } \mathbf{X},$$

$$\widetilde{\mathbf{X}} = \text{non-function (template) declarations in scope } \mathbf{X}, \text{ and}$$

$$\mathbf{ADL} = \text{associated namespaces of reference's arguments}.$$

ADL Scoping Rules

when scope **H** is an inner namespace scope

$$\boxed{\mathbf{fn_{ADL}}\,(\mathbf{H}) \lhd \Lbag_{i=1}^{n-1} \mathbf{fn_{ADL}}\,(\mathbf{N_i}) \lhd \left(\mathbf{N_n} \Leftrightarrow \mathbf{U_{N_n}} \Leftrightarrow \left(\left(\widetilde{\mathbf{N}}_\mathbf{n} \Leftrightarrow \widetilde{\mathbf{U}}_{\mathbf{N_n}}\right) \lhd \mathbf{ADL}\right)\right)}$$

$$\mathrm{fn}_{\mathrm{ADL}}\,(\mathrm{N}) = \mathrm{N} \Leftrightarrow \mathrm{U_N} \Leftrightarrow \left(\left(\mathrm{N} \Leftrightarrow \mathrm{U_N}\right) \rhd \left(\left(\widetilde{\mathrm{N}} \Leftrightarrow \widetilde{\mathrm{U}}_\mathrm{N}\right) \lhd \mathrm{ADL}\right)\right),$$

$\mathbf{N_1} \cdots \mathbf{N_n}$ = surrounding namespaces,

$\mathbf{H}$ = scope where name binding is triggered from,

$\mathbf{U_X}$ = using declarations in scope $\mathbf{X}$,

$\widetilde{\mathbf{X}}$ = non-function (template) declarations in scope $\mathbf{X}$, and

$\mathbf{ADL}$ = associated namespaces of reference's arguments.

ADL Scoping Rules

when scope **H** is the outermost namespace scope

$$\left(\mathbf{N_n} \Leftrightarrow \mathbf{U_{N_n}} \Leftrightarrow \left(\left(\widetilde{\mathbf{N}}_\mathbf{n} \Leftrightarrow \widetilde{\mathbf{U}}_{\mathbf{N_n}}\right) \lhd \mathbf{ADL}\right)\right)$$

$\mathbf{H} = \mathbf{N}_\mathbf{n}$

$$\begin{aligned} \mathbf{H} &= \text{scope where name binding is triggered from,} \\ \mathbf{U_X} &= \text{using declarations in scope } \mathbf{X}, \\ \widetilde{\mathbf{X}} &= \text{non-function (template) declarations in scope } \mathbf{X}, \text{ and} \\ \mathbf{ADL} &= \text{associated namespaces of reference's arguments.} \end{aligned}$$

C++ Operators Example

```
struct X {}; struct Y {};

void operator+(X, X) { }
void operator+(X, Y) { }

void test(X x, Y y) {
  void operator+(X, X);
  x + x;
  x + y;
  operator+(x, x);
  operator+(x, y);
}
```

test

::

$$(\textbf{test} \Leftrightarrow \textbf{builtin-ops} \Leftrightarrow \textbf{X}) \lhd ::$$

C++ Operators Scoping Rules

without ADL

$$\left(\mathbf{H} \Leftrightarrow \mathbf{U_H} \Leftrightarrow \boxed{\mathbf{O_B} \Leftrightarrow \mathbf{M}}\right) \lhd \Lbag_{i=1}^{s} \mathbf{S_i} \lhd \Lbag_{i=1}^{n} \mathbf{N_i}$$

$$\begin{aligned}
\mathbf{S_1} \cdots \mathbf{S_s} &= \text{surrounding non-namespace scopes,} \\
\mathbf{N_1} \cdots \mathbf{N_n} &= \text{surrounding namespaces,} \\
\mathbf{H} &= \text{scope where name binding is triggered from,} \\
\mathbf{O_B} &= \text{built-in operators,} \\
\mathbf{M} &= \text{member scope of operator's first argument,} \\
\mathbf{U_X} &= \text{using declarations in scope } \mathbf{X}\text{, and}
\end{aligned}$$

C++ Operators Scoping Rules

with ADL

$$\left(\mathbf{H} \Leftrightarrow \mathbf{U_H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M} \Leftrightarrow \left(\mathbf{U_H} \rhd \left(\left(\boxed{\mathbf{H}} \Leftrightarrow \boxed{\widetilde{\mathbf{U}}_\mathbf{H}} \Leftrightarrow \boxed{\mathbf{O_B} \Leftrightarrow \mathbf{M}}\right) \lhd \mathbf{ADL}\right)\right)\right)$$
$$\lhd \Lbag_{i=1}^{s} \boxed{\mathbf{f_{ADL}}\left(\mathbf{S_i}\right)} \lhd \Lbag_{i=1}^{n-1} \boxed{\mathbf{fn_{ADL}}\left(\mathbf{N_i}\right)} \lhd \left(\mathbf{N_n} \Leftrightarrow \mathbf{U_{N_n}} \Leftrightarrow \left(\boxed{\left(\widetilde{\mathbf{N}}_\mathbf{n} \Leftrightarrow \widetilde{\mathbf{U}}_{\mathbf{N_n}}\right)} \lhd \mathbf{ADL}\right)\right)$$

$$\mathbf{f_{ADL}}\left(\mathbf{X}\right) = \mathbf{X} \Leftrightarrow \mathbf{U_X} \Leftrightarrow \left(\mathbf{U_X} \rhd \left(\left(\mathbf{X} \Leftrightarrow \boxed{\widetilde{\mathbf{U}}_\mathbf{X}}\right) \lhd \mathbf{ADL}\right)\right),$$

$\mathbf{S_1} \cdots \mathbf{S_s}$ = surrounding non-namespace scopes,

$$\mathbf{fn_{ADL}}\left(\mathbf{N}\right) = \mathbf{N} \Leftrightarrow \mathbf{U_N} \Leftrightarrow \left(\left(\mathbf{N} \Leftrightarrow \mathbf{U_N}\right) \rhd \left(\boxed{\left(\widetilde{\mathbf{N}} \Leftrightarrow \widetilde{\mathbf{U}}_\mathbf{N}\right)} \lhd \mathbf{ADL}\right)\right),$$

$\mathbf{N_1} \cdots \mathbf{N_n}$ = surrounding namespaces,

$\mathbf{H}$ = scope where name binding is triggered from,

$\mathbf{O_B}$ = built-in operators,

$\mathbf{M}$ = member scope of operator's first argument,

$\mathbf{U_X}$ = using declarations in scope $\mathbf{X}$,

$\widetilde{\mathbf{X}}$ = non-function (template) declarations in scope $\mathbf{X}$, and

$\mathbf{ADL}$ = associated namespaces of reference's arguments.

Empty, since `operator` is a reserved keyword.

C++ Operators Scoping Rules

with ADL

$$\left(\mathbf{H} \Leftrightarrow \mathbf{U_H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M} \Leftrightarrow \left(\mathbf{U_H} \triangleright \left(\left(\mathbf{H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M}\right) \triangleleft \mathbf{ADL}\right)\right)\right)$$
$$\triangleleft \llparenthesis_{i=1}^{s} \mathbf{f_{ADL}}\left(\mathbf{S_i}\right) \triangleleft \llparenthesis_{i=1}^{n-1} \mathbf{fn_{ADL}}\left(\mathbf{N_i}\right) \triangleleft \left(\mathbf{N_n} \Leftrightarrow \mathbf{U_{N_n}} \Leftrightarrow \mathbf{ADL}\right)$$

$$\begin{aligned}
\mathbf{f_{ADL}}\left(\mathbf{X}\right) &= \mathbf{X} \Leftrightarrow \mathbf{U_X} \Leftrightarrow \left(\mathbf{U_X} \triangleright \left(\mathbf{X} \triangleleft \mathbf{ADL}\right)\right), \\
\mathbf{S_1} \cdots \mathbf{S_s} &= \text{surrounding non-namespace scopes}, \\
\mathbf{fn_{ADL}}\left(\mathrm{N}\right) &= \mathrm{N} \Leftrightarrow \mathrm{U_N} \Leftrightarrow \left(\left(\mathrm{N} \Leftrightarrow \mathrm{U_N}\right) \triangleright \mathbf{ADL}\right), \\
\mathbf{N_1} \cdots \mathbf{N_n} &= \text{surrounding namespaces}, \\
\mathbf{H} &= \text{scope where name binding is triggered from}, \\
\mathbf{O_B} &= \text{built-in operators}, \\
\mathbf{M} &= \text{member scope of operator's first argument}, \\
\mathbf{U_X} &= \text{using declarations in scope } \mathbf{X}\text{, and} \\
\mathbf{ADL} &= \text{associated namespaces of reference's arguments.}
\end{aligned}$$

C++ Operators Scoping Rules

with ADL

$$\boxed{\begin{gathered}\left(\mathbf{H} \Leftrightarrow \mathbf{U_H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M} \Leftrightarrow \left(\mathbf{U_H} \triangleright \left(\left(\mathbf{H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M}\right) \triangleleft \mathbf{ADL}\right)\right)\right) \\ \triangleleft \Lleftarrow_{i=1}^{s} \mathbf{f_{ADL}}\left(\mathbf{S_i}\right) \triangleleft \left(\Lleftarrow_{i=1}^{n}\left(\mathbf{N_i} \Leftrightarrow \mathbf{U_{N_i}}\right) \Leftrightarrow \mathbf{ADL}\right)\end{gathered}}$$

$$\begin{aligned}\mathbf{f_{ADL}}\left(\mathbf{X}\right) &= \mathbf{X} \Leftrightarrow \mathbf{U_X} \Leftrightarrow \left(\mathbf{U_X} \triangleright \left(\mathbf{X} \triangleleft \mathbf{ADL}\right)\right), \\ \mathbf{S_1} \cdots \mathbf{S_s} &= \text{surrounding non-namespace scopes,}\end{aligned}$$

$$\begin{aligned}\mathbf{N_1} \cdots \mathbf{N_n} &= \text{surrounding namespaces,} \\ \mathbf{H} &= \text{scope where name binding is triggered from,} \\ \mathbf{O_B} &= \text{built-in operators,} \\ \mathbf{M} &= \text{member scope of operator's first argument,} \\ \mathbf{U_X} &= \text{using declarations in scope } \mathbf{X}\text{, and} \\ \mathbf{ADL} &= \text{associated namespaces of reference's arguments.}\end{aligned}$$

C++ Operators Scoping Rules

when scope **H** is an inner namespace scope

$$\left(\mathbf{H} \Leftrightarrow \mathbf{U_H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M} \Leftrightarrow \left(\left(\mathbf{H} \Leftrightarrow \mathbf{U_H}\right) \triangleright \left(\left(\mathbf{O_B} \Leftrightarrow \mathbf{M}\right) \triangleleft \mathbf{ADL}\right)\right)\right)$$
$$\triangleleft \left(\langle\!|_{i=1}^{n} \left(\mathbf{N_i} \Leftrightarrow \mathbf{U_{N_i}}\right) \Leftrightarrow \mathbf{ADL}\right)$$

$\mathbf{N_1} \cdots \mathbf{N_n}$ = surrounding namespaces,
$\mathbf{H}$ = scope where name binding is triggered from,
$\mathbf{O_B}$ = built-in operators,
$\mathbf{M}$ = member scope of operator's first argument,
$\mathbf{U_X}$ = using declarations in scope $\mathbf{X}$, and
$\mathbf{ADL}$ = associated namespaces of reference's arguments.

C++ Operators Scoping Rules

when scope **H** is the outermost namespace scope

$$\Big(\mathbf{H} \Leftrightarrow \mathbf{U_H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M} \Leftrightarrow \Big(\qquad\qquad\qquad \Big(\Big(\mathbf{O_B} \Leftrightarrow \mathbf{M}\Big) \lhd \mathbf{ADL}\Big)\Big)\Big)$$

$$\begin{aligned}
\mathbf{H} &= \text{scope where name binding is triggered from,}\\
\mathbf{O_B} &= \text{built-in operators,}\\
\mathbf{M} &= \text{member scope of operator's first argument,}\\
\mathbf{U_X} &= \text{using declarations in scope } \mathbf{X}\text{, and}\\
\mathbf{ADL} &= \text{associated namespaces of reference's arguments.}
\end{aligned}$$

2. Executing Scoping Rules

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \rhd (\mathbf{test} \lhd \mathbf{adl}))) \lhd (\mathbf{::} \Leftrightarrow \mathbf{adl})$$

1. Use scope combinators.
2. Instantiate the **Language** concept.

```
void foo();
namespace adl { struct X {};
                void foo(X); }
namespace ns { void foo(int); }
void test(adl::X x) {
    using ns::foo;
    foo(x);
}
```

Bind

- **Best viable candidate, or**
- **error**

$$\mathbf{bind} : \mathbf{Ref} \times \mathbf{Scope}_{\mathbf{Ref},\mathbf{Decl}} \rightarrow (\mathbf{Decl} + \mathrm{Error})$$

Elementary Name Binding

Scopes as sets of declarations.

```
lookup :: (…) => ref -> Set decl -> OverloadSet decl
lookup ref decls = Set.filter (match ref) decls
```

```
resolve :: (…) => ref -> OverloadSet decl -> Maybe decl
resolve ref decls = assess $ select_best_viable ref decls
```

```
assess :: (…) => BestViableSet decl -> Maybe decl
assess decls = case (Set.elems decls) of
                 []     -> Nothing
                 [decl] -> (Just decl)
                 _      -> ambiguity decls
```

Elementary Name Binding

- **What is the necessary minimal abstraction?**

```
lookup :: (…) => ref -> Set decl -> OverloadSet decl
lookup ref decls = Set.filter (match ref) decls
```

```
resolve :: (…) => ref -> OverloadSet decl -> Maybe decl
resolve ref decls = assess $ select_best_viable ref decls
```

```
assess :: (…) => BestViableSet decl -> Maybe decl
assess decls = case (Set.elems decls) of
                 []     -> Nothing
                 [decl] -> (Just decl)
                 _      -> ambiguity decls
```

An Abstraction of PLs

- The **`Language`** concept:
  - `match`: How references match declarations.
  - `select_best_viable`: How best viable candidates are determined.
  - `ambiguity`: How to handle ambiguous results.

An Abstraction of PLs

- The **`Ambiguity`** concept:
  - `ambiguity`: How to handle ambiguous results.
- The **`Language`** concept:
  - `match`: How references match declarations.
  - `select_best_viable`: How best viable candidates are determined.
  - refines **`Ambiguity`**.

An Abstraction of PLs, Ext'd

- The **`Ambiguity`** concept:
  - `ambiguity`: How to handle ambiguous results.
- The **`Language`** concept:
  - `match`: How references match declarations.
  - `select_best_viable`: How best viable candidates are determined.
  - refines **`Ambiguity`**.
- The **`Parameterized`** concept:
  - expresses changes in the bind environment during name binding.
  - e.g., **changes in `ambiguity`** for a variant of ***weak hiding***.

An Abstraction of PLs

```
class Ambiguity decl where
  ambiguity :: ViableSet decl -> Maybe decl
  ambiguity _ = Nothing                                  -- default.
```

```
class Ambiguity decl => Language ref decl where
  match :: ref -> decl -> bool
  select_best_viable :: ref -> OverloadSet decl -> BestViableSet decl
```

An Abstraction of PLs, Ext'd

```
class Ambiguity decl where
  ambiguity :: ViableSet decl -> Maybe decl
  ambiguity _ = Nothing                                 -- default.
```

```
class Ambiguity decl => Basic.Language ref decl where
  match :: ref -> decl -> bool
  select_best_viable :: ref -> OverloadSet decl -> BestViableSet decl
```

```
class (Parameterized decl, Basic.Language ref decl) =>
       Language ref decl
```

Elementary Name Binding

```
lookup :: (Language ref decl) => ref -> Set decl -> OverloadSet decl
lookup ref decls = Set.filter (match ref) decls
```

```
resolve :: (Language ref decl) => ref -> OverloadSet decl -> Maybe decl
resolve ref decls = assess $ select_best_viable ref decls
```

```
assess :: (Ambiguity decl) => BestViableSet decl -> Maybe decl
assess decls = case (Set.elems decls) of
                  []     -> Nothing
                  [decl] -> (Just decl)
                  _      -> ambiguity decls
```

- `assess` requires the **`Ambiguity`** concept only,
  gets the meaning of ambiguity from **the language**.

Elementary Name Binding, Ext'd

```
resolve :: (Basic.Language ref decl) =>
           BindEnv decl -> ref -> OverloadSet decl -> Maybe decl
resolve env ref decls = assess env $ select_best_viable ref decls
```

```
assess :: BindEnv decl -> BestViableSet decl -> Maybe decl
assess env decls = case (Set.elems decls) of
                     []     -> Nothing
                     [decl] -> (Just decl)
                   _        -> ambiguity env decls
```

- `assess` requires **no** concept,

  gets the meaning of ambiguity from **the bind environment**.

CENTER FOR RESEARCH IN EXTREME SCALE TECHNOLOGIES
INDIANA UNIVERSITY
Pervasive Technology Institute

Applications

- Understanding current name binding mechanisms:
  - Argument-dependent lookup.
  - C++ operators.
  - A cross-language analysis.
- Exploring concepts designs
  - Understanding the current limitations.
  - Exploring new solutions:
    - weak hiding,
    - 2-Stage name binding, and
    - parameterized weak hiding.
- Simplifying compiler designs.

A cross language analysis

- Instantiate the **`Language`** concept with different languages, designs and kinds of references:
  - C++: function calls
    - **`match`** = have_**same_name** + **filter** out types.
    - **`select_best_viable`** = overload resolution (incl. viability check).
    - **`ambiguity`** = ambiguity_**is_error**.
  - C++: uses of types
    - **`match`** = have_**same_name**.
    - **`select_best_viable`** = viability check if single result, identity otherwise.
    - **`ambiguity`** = ambiguity_**is_error**.
  - C++: Multimethods proposal (cf. N2216)
    - **`match`** = (same as function calls).
    - **`select_best_viable`** = (same as function calls).
    - **`ambiguity`** = ambiguity_**is_not_error**.

C++ Multimethods (Std. Doc. N2216)

titled "Open Multi-Methods for C++", Pirkelbauer et al., 2007

- Relaxes the rules for ambiguity,
  - allowing ambiguous calls when all best viable candidates "have **a unique-base method** through which the call can be dispatched".

```
struct X, Y, Z;
void foo(virtual X&, virtual Y&); // (1)
void foo(virtual Y&, virtual Y&); // (2) – ** unique-base method **
void foo(virtual Y&, virtual Z&); // (3)
struct XY : X, Y {}
struct YZ : Y, Z {}
void foo(virtual XY&, virtual Y&); // (4) – overrider for (1) and (2)
void foo(virtual Y&, virtual YZ&); // (5) – overrider for (2) and (3)

XY xy; YZ yz;
foo(xy,yz);     // both (4) and (5) are equally viable matches,
                //  w/ unique base (2)
```

A cross language analysis

- Instantiate the **`Language`** concept with different languages, designs and kinds of references:
  - Haskell: name uses, excluding type inference
    - **`match`** = have_**same_name**.
    - **`select_best_viable`** = identity.
    - **`ambiguity`** = ambiguity_**is_error**.
  - Haskell: name uses, including type inference
    - **`match`** = have_**same_name**.
    - **`select_best_viable`** = viability check if single result, identity otherwise.
    - **`ambiguity`** = ambiguity_**is_error**.
  - Alternatives have different implications with weak hiding or **Bind**^{x2}.

Applications

❖ Understanding current name binding mechanisms:

- Argument-dependent lookup.
- C++ Operators.
- A cross-language analysis.

❖ Exploring concepts designs

- Understanding the current limitations.
- Exploring new solutions:
  - weak hiding,
  - 2-Stage name binding, and
  - parameterized weak hiding.

❖ Simplifying compiler designs

Problem: Current Limitations

❖ Current scoping rules break seemingly valid codes.

::

```
concept Foo2<typename P> =
   requires (P a, P b) { foo(a, b); }

void foo(int) { }

template<Foo2 T>
void gen_func(T a, T b) {
   foo(a, b);
   foo(1);
}
```

Foo2<T>

Should Foo2<T>::foo() shadow ::foo()? How?

Reject or accept the call foo(1)?

Practical Examples

- STL: `rotate()` and `move()`
  - Two function declarations, different number of parameters.
  - Name lookup only finds one, type-check fails.
- Plenoptic photography: Image rendering
  - Two function declarations, different parameter types.
  - Name lookup only finds one, type-check fails.
- STL: `common_type`
  - Two type (function) declarations, different type parameters.
  - Name lookup only finds one, type-check fails.

STL: rotate() and move()

```
// Specialization adapted from latest release of libstdc++
template<RandomAccessIterator I>
  requires Permutable<I>       // has move(ValueType<I>&&)
I rotate (I first, I middle, I last) {
   …
   if (__is_pod(ValueType<I>) && (middle - first == 1)) {
      ValueType<I> t = std::move(*p);            //
      std::move(p+1, p+last-first, p);           //
      *(p + last – first -1) = std::move(t);  //
   }            // Ok w/ constrained std::move(), But…
   …            // Does not use the constraints.
}
```

CENTER FOR RESEARCH IN EXTREME SCALE TECHNOLOGIES
INDIANA UNIVERSITY
Pervasive Technology Institute

STL: rotate() and move()

```
// A novice's approach
template<RandomAccessIterator I>
  requires Permutable<I>        // has move(ValueType<I>&&)
I rotate (I first, I middle, I last) {
    …
    if (__is_pod(ValueType<I>) && (middle - first == 1)) {
        ValueType<I> t = move(*p);              // Ok
        move(p+1, p+last-first, p);             // …
        *(p + last – first -1) = move(t);       // Ok
    }      // Compile error… or… Possible ADL issues…
    …
}
```

STL: rotate( ) and move( )

```
// The right way…
template<RandomAccessIterator I>
  requires Permutable<I>  // has move(ValueType<I>&&)
I rotate (I first, I middle, I last) {
   …
   if (__is_pod(ValueType<I>) && (middle - first == 1)) {
      ValueType<I> t = move(*p);                // Ok
      std::move(p+1, p+last-first, p);          // Ok
      *(p + last – first -1) = move(t);         // Ok
   }     // Shadow or No Shadow… But…
   …     // Requires change in existing implementation.
}
```

STL: rotate() and move()

```
// Update MoveWritable concept …
template<RandomAccessIterator I>
  requires Permutable<I>  // has std::move(ValueType<I>&&)
I rotate (I first, I middle, I last) {
    …
    if (__is_pod(ValueType<I>) && (middle - first == 1)) {
        ValueType<I> t = std::move(*p);              // Ok
        std::move(p+1, p+last-first, p);             //
        *(p + last – first -1) = std::move(t);   // Ok
    }        // Compile error… or… No ADL issues…
    …           // Maybe OK to not shadow?
}             // std::move() is not a customization point.
```

Plenoptic photo.: Image rendering

```
template<typename PixelType>
struct Radiance {
  typedef typename
    boost::multi_array<PixelType, 4> RadianceType;
  RadianceType pixels;     // 4D array of pixels
  …
  void Read(const string &ImageFile, …);
  Radiance<PixelType> Render_Basic(…);
  Radiance<PixelType> Render_Blended(…);
  … // Several rendering variants …
  void Print(const string& OutputFile, …);
  …
}
```

Plenoptic photo.: Image rendering

```
template<typename PixelType>
Radiance<PixelType> Radiance<PixelType>::Render_Blended(…) {
  …
  RadianceType Rendered(boost::extents[Iy][Ix][1][1]);  …
  for (…) { …                                 // for each image pixel
    PixelType pixel_avg; …
    for (…) { …                               // for each direction
      pixel_avg += pixels[ri][rj][rl][rm]; … // integrate pixel
    } …
    Rendered[i][j][0][0] = move(pixel_avg); …
  }
  return move(new Radiance<PixelType>(move(Rendered), Ix, Iy));
}
```

Plenoptic photo.: Image rendering

```
template<typename PixelType>
using MultiArrayIterator =
      boost::multi_array<PixelType, 4>::iterator

// Add safety
template<Regular PixelType>
  requires IndirectlyCopyable<MultiArrayIterator<PixelType>,
                              MultiArrayIterator<PixelType>>
struct Radiance {
  typedef typename
    boost::multi_array<PixelType, 4> RadianceType;
  …
}
```

Plenoptic photo.: Image rendering

```
template<Regular PixelType> …
Radiance<PixelType>&& Radiance<PixelType>::Render_Blended(…) {
  …
  RadianceType Rendered(boost::extents[Iy][Ix][1][1]);  …
  for (…) { …                                  // for each image pixel
    PixelType pixel_avg; …
    for (…) { …                                // for each direction
      pixel_avg += pixels[ri][rj][rl][rm]; … // integrate pixel
    } …
    Rendered[i][j][0][0] = move(pixel_avg); …     // Ok.
  }          // Compile error.
  return move(Radiance<PixelType>(move(Rendered), Ix, Iy));
}
```

STL: common_type

```
concept Common<typename T, typename U> =
  requires { common_type<T, U>::type;    axiom (…) { … } };

template<typename T, typename U>
requires Common<T, U> && …
void gen_func(…) {
    common_type<T, U>::type …                  // Ok.
    common_type<int>::type …                   // Error.
    common_type<int, double>::type …           // Error.
    common_type<int, double, char>::type …  // Error.
}
```

Solution: Weak Hiding

::

```
concept Foo2<typename P> =
    requires (P a, P b) { foo(a, b); }

void foo(int) { }

template<Foo2 T>
void gen_func(T a, T b) {
    foo(a, b);
    foo(1);
}
```

Foo2<T>

Foo2<T>::foo() should weakly hide ::foo()!

Accept the call foo(1)!

The Weak Hiding Scoping Rule

$$\mathrm{s_1} \Lleftarrow \mathrm{s_2} = \left(\mathit{resolve}_{ref} \circ \mathit{lookup}_{ref}\right) \mathrm{s_1} \;?\; \mathit{lookup}_{ref}\, \mathrm{s_1} \;:\; \mathit{lookup}_{ref}\, \mathrm{s_2}$$

```
concept Foo2<typename P> =
    requires (P a, P b) { … }

void foo(int) { }

template<Foo2 T>
void gen_func(T a, T b) {
    foo(a, b);
    foo(1);
}
```

Foo2<T>

::

$$\text{gen\_func} \lhd \text{P} \lhd (\textbf{Concept<T>} \Leftrightarrow \text{T}) \Lleftarrow \textbf{::}$$

Result:

Binds **foo()** to **::foo()**.

Implementing Weak Hiding

$$s_1 \Lleftarrow\!\mid s_2 = \boxed{\left(\boldsymbol{resolve}_{ref} \circ \boldsymbol{lookup}_{ref}\right) s_1}\ ?\ \boldsymbol{lookup}_{ref}\, s_1 \ :\ \boldsymbol{lookup}_{ref}\, s_2$$

$$\boldsymbol{bind}\ (ref, scope) = \boxed{((\boldsymbol{resolve}\ ref) \circ (\boldsymbol{lookup}\ ref))}\ scope$$

- Implementation = **Two-Stage Name Binding (Bind^{x2})**
  1. Bind with inner scope: **s1**.
  2. Bind with outer scope: **s2**.

- Bind^{x2} repeats name binding under different contexts.

Bind^{x2} for C++ Concepts

1. **Within restricted scope**:
   - up to the outermost restricted scope.
   - Disables ADL and some qualified name lookups.
2. **In surrounding scope**:
   - normal lookup – including ADL.

```
concept Foo2<typename P> =
    requires (P a, P b) { … }

void foo(int) { }

template<Foo2 T>
void gen_func(T a, T b) {
    foo(a, b);
    foo(1);
}
```

Foo2<T>

::

Applications

- Understanding current name binding mechanisms:
  - Argument-dependent lookup.
  - C++ Operators.
  - A cross-language analysis.

- Exploring concepts designs
  - Understanding the current limitations.
  - Exploring new solutions:
    - weak hiding,
    - 2-Stage name binding, and
    - parameterized weak hiding.

- Simplifying compiler designs

Inner Scope Ambiguity

when ambiguity IS an error

```
concept Foo2<typename P> {
    void foo(P, int);
    void foo(int, P);
}

template<Foo2 T>
void gen_func(T a, int b) {
    foo(b, b);
}
```

Current C++ Concepts:

Reject!

Inner Scope Ambiguity

when ambiguity IS an error

```
concept Foo2<typename P> {
    void foo(P, int);
    void foo(int, P);
}

template<Foo2 T>
void gen_func(T a, int b) {
    foo(b, b);
}
```

Proposed Extension:

Accept!

Inner Scope Ambiguity

when ambiguity IS an error

False

Repeat `bind`

$$\mathbf{s_1} \Lleftarrow \mathbf{s_2} = \left(\boldsymbol{resolve}_{ref} \circ \boldsymbol{lookup}_{ref}\right)\ \mathbf{s_1}\ ?\ \boldsymbol{lookup}_{ref}\ \mathbf{s_1}\ :\ \boldsymbol{lookup}_{ref}\ \mathbf{s_2}$$

$$\boldsymbol{bind}\ (ref, scope) = ((\boldsymbol{resolve}\ ref) \circ (\boldsymbol{lookup}\ ref))\ scope$$

Inner Scope Ambiguity

when ambiguity IS an error

False

Repeat **bind**

$$s_1 \Lleftarrow\!\mid s_2 = (resolve_{ref} \circ lookup_{ref})\ s_1\ ?\ lookup_{ref}\ s_1 : lookup_{ref}\ s_2$$

```
concept Foo2<typename P> {
    void foo(P, int);
    void foo(int, P);
}

template<Foo2 T>
void gen_func(T a, int b) {
    foo(b, b);
}
```

Not Current C++ Concepts:

Reject or bind to unintended!

Inner Scope Ambiguity

when ambiguity IS an error

True

temporary change in ambiguity

Do not repeat `bind`

$$s_1 \Lleftarrow s_2 = \left(\mathit{resolve}_{ref} \circ \left(\mathit{update} \circ \mathit{lookup}_{ref}\right)\right)\ s_1\ ?\ \mathit{lookup}_{ref}\ s_1 : \mathit{lookup}_{ref}\ s_2$$

```
concept Foo2<typename P> {
    void foo(P, int);
    void foo(int, P);
}

template<Foo2 T>
void gen_func(T a, int b) {
    foo(b, b);
}
```

Current C++ Concepts:
Reject!

Inner Scope Ambiguity

when ambiguity IS NOT an error

True

Do not repeat `bind`

$$s_1 \Lleftarrow s_2 = (resolve_{ref} \circ lookup_{ref})\ s_1\ ?\ lookup_{ref}\ s_1 : lookup_{ref}\ s_2$$

```
concept Foo2<typename P> {
    void foo(P, int);
    void foo(int, P);
}

template<Foo2 T>
void gen_func(T a, int b) {
    foo(b, b);
}
```

Not Current C++ Concepts:

Accept!

Inner Scope Ambiguity

when ambiguity IS NOT an error

False

Repeat bind

$$\mathbf{s_1} \Lleftarrow\!\!| \ \mathbf{s_2} = \boldsymbol{\left(resolve_{ref} \circ lookup_{ref}\right)}\ \mathbf{s_1}\ \mathbf{?}$$
$$\boldsymbol{lookup_{ref}}\ \mathbf{s_1}\ \mathbf{:}\ \boldsymbol{lookup_{ref}}\ \mathbf{s_2}$$

```
concept Foo2<typename P> {
    void foo(P, int);
    void foo(int, P);
}

template<Foo2 T>
void gen_func(T a, int b) {
    foo(b);
}
```

Not Current C++ Concepts:
Accept unintended ambiguity!

Ambiguity for C++ Concepts:

What is the most desirable?

1. Ambiguity IS an error, always?
2. Ambiguity IS NOT an error, always?
3. **A middle ground option?**

Rejects desirable, or binds to undesirable

Accepts undesirable

```
concept Foo2<typename P> {
    void foo(P, int);
    void foo(int, P);
}

template<Foo2 T>
void gen_func(T a, int b) {
    foo(b, b); foo(b);
}
```

Proposed Extension:

Accept only desirable! (?)

Ambiguity for C++ Concepts

Our proposed extension

- Ambiguity IS NOT an error, when in restricted scope.
  - Similar to multimethods proposal N2216.
- Ambiguity IS an error, otherwise.

```
concept Foo2<typename P> {
    void foo(P, int);
    void foo(int, P);
}

template<Foo2 T>
void gen_func(T a, int b) {
    foo(b, b); foo(b);
}
```

Proposed Extension:
Accept only desirable!

Revisiting Hinnant's Example

- Cf. C++ standard documents:
  - C++ Library Reflector message c++std-lib-20050
  - N2576: *"Type-Soundness and Optimization in the Concepts Proposal"*, Douglas Gregor, March 2008.
- Question: **How to type-check constrained templates?**

```
template <class T>
    requires std::DefaultConstructible<T> && std::Addable<T> &&
    std::Convertible<T::result_type,T> && std::CopyConstructible<T>
void test() {
    T s1;
    T s2;
    T s3 = s1 + s2 + T() + T() + T();
}

int main() { test<string>(); }
```

Hinnant's Example: Problem

- The desired (optimized) output
  - is observed with unconstrained templates, but
  - is not observed with constrained templates.

```
auto concept Addable<typename T, typename U=T> {
    typename result_type;
    result_type operator+(T const&, U const&);
}

template <class T>
    requires … std::Addable<T> …
void test() {
    T s1;
    T s2;
    T s3 = s1 + s2 + T() + T() + T();
}

int main() { test<string>(); }
```

Unconstrained:

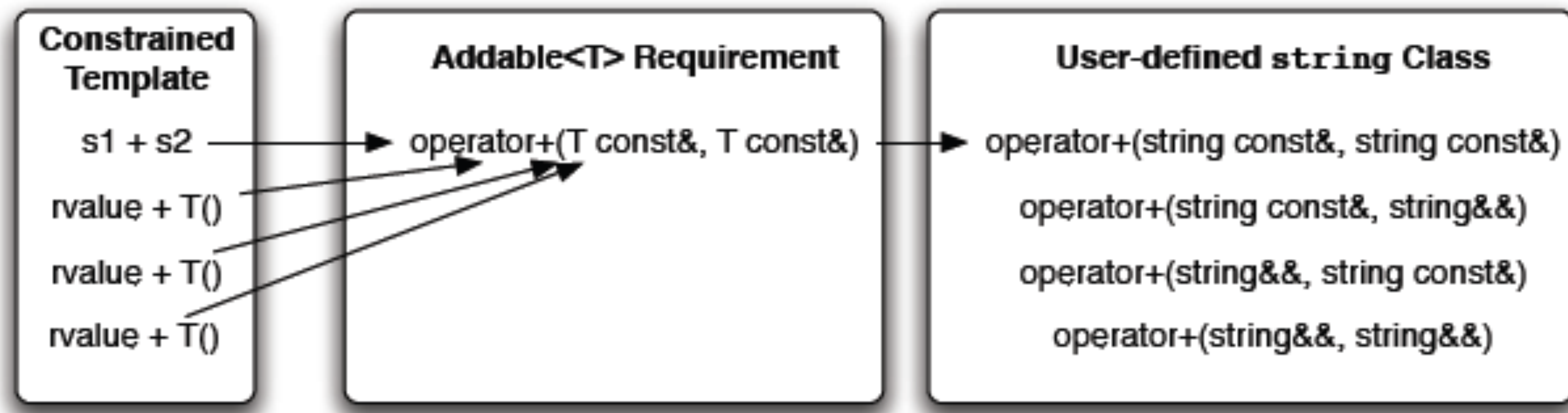
lv string + lv string
rv string += rv string
rv string += rv string
rv string += rv string

Constrained:

lv string + lv string
lv string + lv string
lv string + lv string
lv string + lv string

Hinnant's Example in Pictures

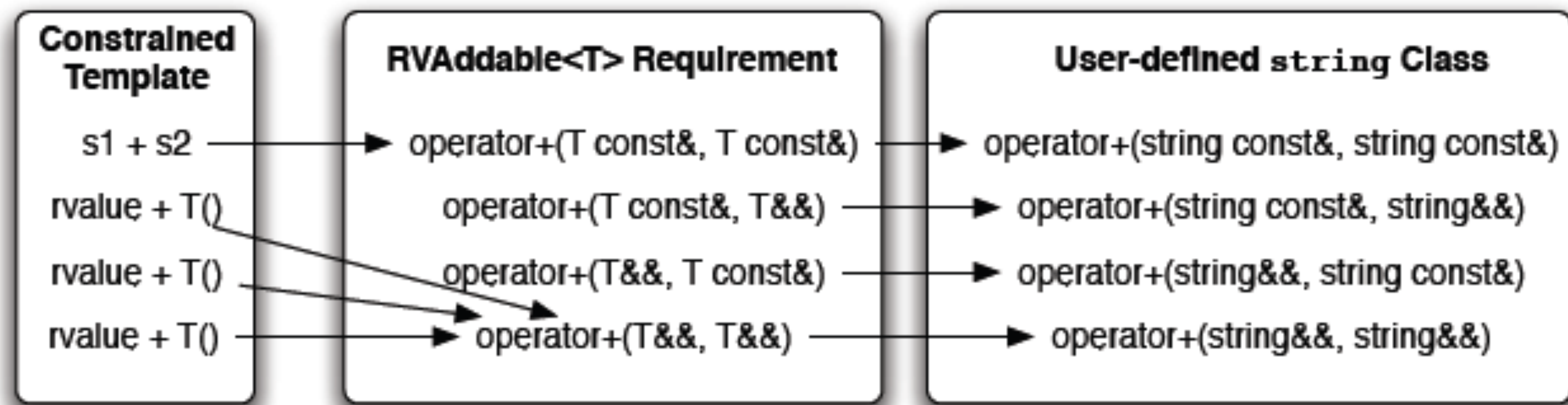
```
auto concept Addable<typename T, typename U = T> {
    typename result_type;
    result_type operator+(T const&, U const&);
}
```



Hinnant's Example: Solution 1

Manual Introduction of Overloads

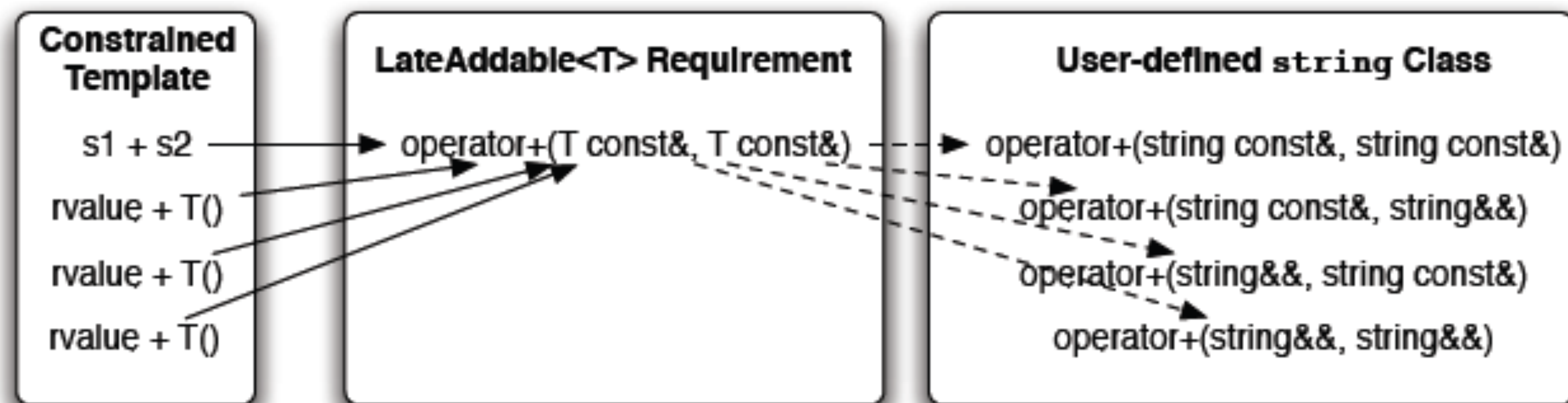
```
auto concept RVAddable<typename T, typename U = T> {
    typename result_type;
    result_type operator+(T const&, U const&);
    result_type operator+(T const&, U&&);
    result_type operator+(T&&, U const&);
    result_type operator+(T&&, U&&);
}
```



Hinnant’s Example: Solution 2

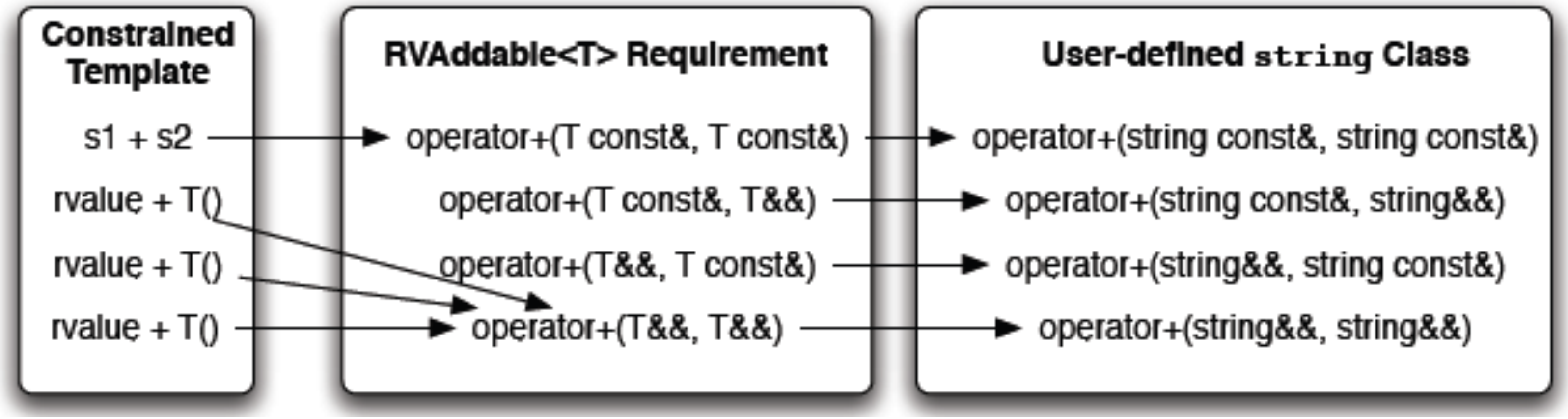
“Eliminating” Forwarding Functions

```
auto concept LateAddable<typename T, typename U = T> {
    typename result_type;
    result_type operator+(T const&, U const&);
}
concept_map LateAddable<string> {
    typedef string result_type;
    string operator+(T const& x, T const& y) { return x + y; }
}
```

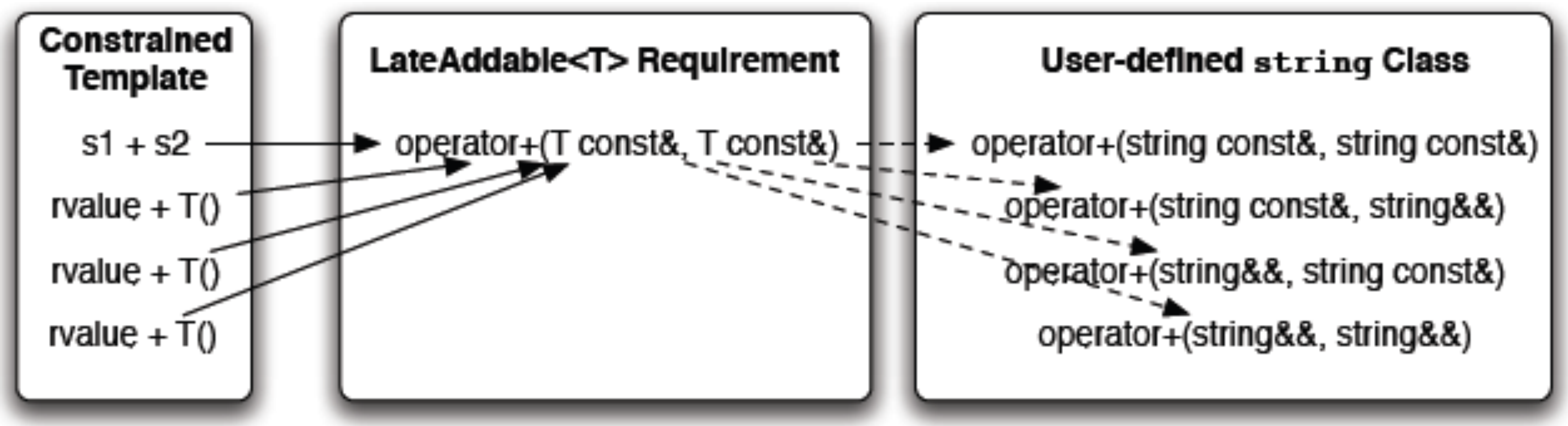


Revisiting the Proposed Solutions

1. Manual introduction of overloads:

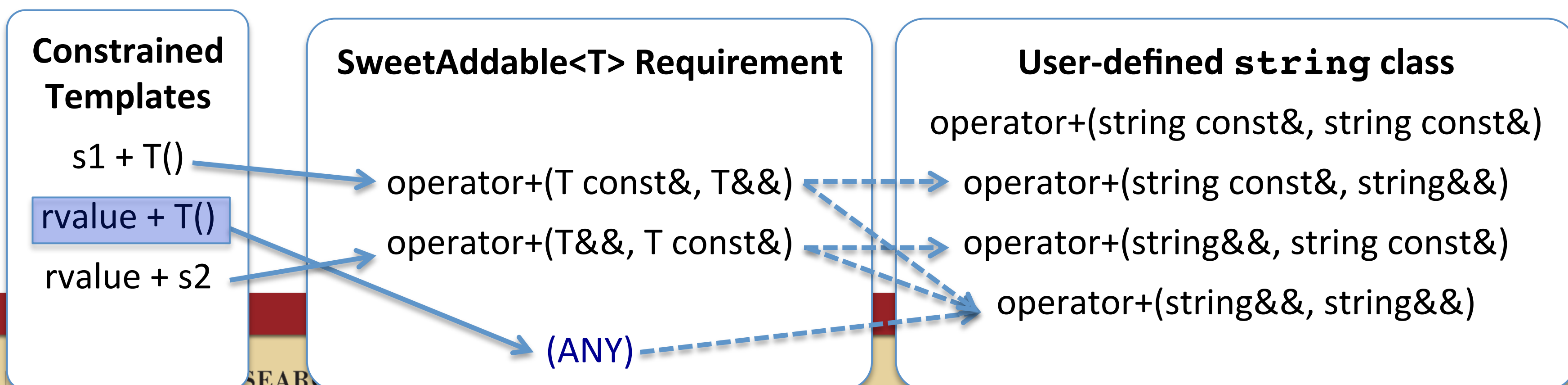


2. “Eliminating” forwarding functions:



A “Sweet Middle” Alternative?

```
auto concept SweetAddable<typename T, typename U = T> {
    typename result_type;
    result_type operator+(T const&, U&&);
    result_type operator+(T&&, U const&);
}
concept_map SweetAddable<string> {
    typedef string result type;
    result_type operator+(T const& x, U&& y) { return x + y; }
    result_type operator+(T&& x, U const& y) { return x + y; }
}
```



A “Sweet Middle” Example

Minimizing and clarifying template constraints

```
template <class T>
    requires … std::SweetAddable<T> …
void test() {
    T s1;
    T s2;
    T s3 = s1 + T() + T() + s2;
}

int main() { test<string>(); }
```

Constrained Templates

s1 + T()

rvalue + T()

rvalue + s2

SweetAddable<T> Requirement

operator+(T const&, T&&)

operator+(T&&, T const&)

(ANY)

User-defined `string` class

operator+(string const&, string const&)

operator+(string const&, string&&)

operator+(string&&, string const&)

operator+(string&&, string&&)

Current C++ Concepts: Problem

- Clarity is penalized.
  - Minimal requirement specification leads to **ambiguity**,
  - which **is an error** in a constrained template definition,
  - even though **instantiation guarantees single matches**.

- Limited control over candidate sets.
  - Either **one or all** possible candidates.

- The “sweet middle” alternative is not allowed.

C++ Concepts: Proposed Extension

- **Ambiguity is not an error**, when in restricted scope.

- Ambiguity remains as defined in the language, otherwise.

→ **Parameterize weak hiding over the bind environment**.

The Combinators, Revisited

- Weak hiding selectively updates lookup results

$$\mathbf{s_1} \Lleftarrow\!| \; \mathbf{s_2} = \left(\mathit{resolve}_{\mathit{ref}} \circ \left(\boxed{\mathit{update} \circ} \, \mathit{lookup}_{\mathit{ref}}\right)\right) \mathbf{s_1} \; ? \; \left(\boxed{\mathit{update} \circ} \, \mathit{lookup}_{\mathit{ref}}\right) \mathbf{s_1} : \mathit{lookup}_{\mathit{ref}} \, \mathbf{s_2}$$

- Define ***update*** based on a new concept: **`Parameterized`**.

```
update :: (Parameterized decl) => OverloadSet decl -> OverloadSet decl
update (os, env) = case get_env of
                       Nothing         -> (os, env)
                       (Just new_env) -> (os, new_env)
```

Lookup results are coupled with the bind environment.

Parameterized example

Bind environment

```
class Parameterized decl where
  get_env :: Maybe (ViableSet decl -> Maybe decl)
  get_env = Nothing                       -- default, constant bind env.
```

```
class (Ambiguity decl, Parameterized decl,
       Basic.Language ref decl) => Language ref decl
```

```
instance Ambiguity decl where
    ambiguity = ambiguity_is_error

instance Parameterized decl where
  get_env = Just (ambiguity_is_not_error)
```

For C++ Concepts

Exploring Concepts Designs: Recap

- Introducing a new scoping rule: **Weak Hiding**.
- Introducing a new mechanism for name binding: **Bind**$^{\mathbf{x2}}$.
- Main Motivation:
  - Software compatibility
  - In transition from unconstrained to constrained polymorphism.
    - e.g., a transition from C++ to ConceptC++.

Exploring Concepts Designs: Recap

- **Bind^{x2}** is an implementation of **weak hiding, optionally parameterized** (e.g., over the **bind environment**).

- Design variants: (Experimental)
  - Constant meaning of ambiguity: from **`Ambiguity`** concept.
  - Changing meaning of ambiguity: from **bind environment**.
    - e.g., Relaxing restrictions for name binding in restricted scope.

Applications

- Understanding current name binding mechanisms:
  - Argument-dependent lookup.
  - C++ Operators.
  - A cross-language analysis.
- Exploring concepts designs
  - Understanding the current limitations.
  - Exploring new solutions:
    - weak hiding,
    - 2-Stage name binding, and
    - parameterized weak hiding.
- Simplifying compiler designs

Using our Framework in Compilers

Option #1: For all combinators, integrate directly into parsers

$$(\textbf{test} \Leftrightarrow \textbf{ns} \Leftrightarrow (\textbf{ns} \rhd (\textbf{test} \lhd \textbf{adl}))) \lhd (\textbf{::} \Leftrightarrow \textbf{adl})$$

1. A **parser interface**: manages scopes and declarations.
2. Instantiate the **Language** concept.

```
void foo();
namespace adl { struct X {};
                void foo(X); }
namespace ns { void foo(int); }
void test(adl::X x) {
    using ns::foo;
    foo(x);
}
```

Bind

- **Best viable candidate, or**
- **error**

Using our Framework in Compilers

Option #2: for weak hiding only, a conservative integration

```
concept Foo2<typename P> =
    requires (P a, P b) { … }
void foo(int) { }
template<Foo2 T>
void gen_func(T a, T b) {
    foo(a, b);
    foo(1);
}
```

Bind^{x2}

- Best viable candidate, or
- error

Prototypes Underway

[https://github.iu.edu/lvoufo/BindIt]

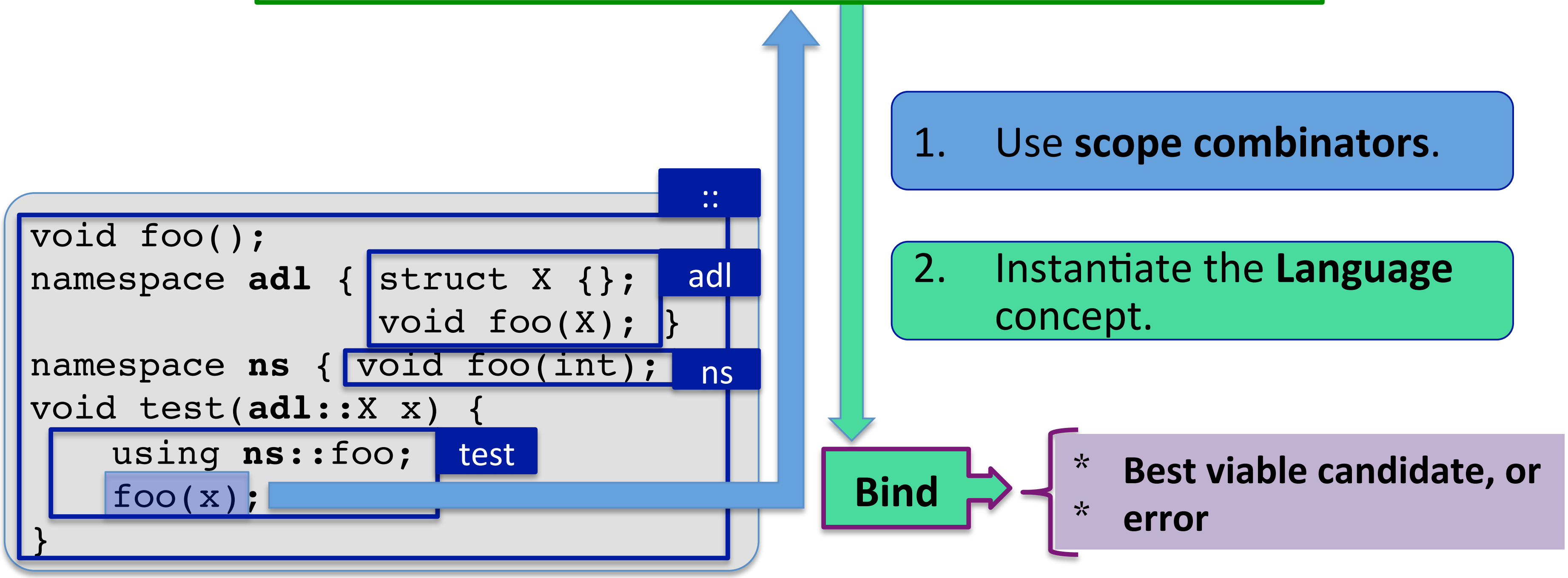
1. Parser interface for managing scopes:
   - Using Haskell's **Parsec** library and a **mini-C++** language.
   - Modifying existing compilers: **language-c**, **featherweight Java**, etc...
2. Executing **Bind**^{x2}:
   - Explored in **ConceptClang**.
3. Alternative compositional view of name binding
   - as composed of some **`lookup_best`** and **`assess`**.

Recap

- Our name binding framework allows:
  - expressing the scoping rules of a language, for a given reference, in terms of 3-4 **scope combinators**, and $\lhd\!| \; , \; \Leftrightarrow \; , \; |\!\rhd \; , \; \Leftarrow\!|$
  - reasoning about the application of scoping rules generically, abstracting over the **`Language`** concept
    - incl. the **`Ambiguity`** and, optionally, **`Parameterized`** concepts.
- 2-stage name binding (**Bind^{x2}**):
  - is an implementation of **weak hiding**, optionally parameterized by the bind environment, and
  - preserves valid programs in transition from C++ to ConceptC++.

Thank You! Questions/Comments?

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \rhd (\mathbf{test} \lhd \mathbf{adl}))) \lhd (\mathbf{::} \Leftrightarrow \mathbf{adl})$$



[https://github.iu.edu/lvoufo/BindIt]