ConceptClang: Theoretical Advances with Full C++ Concepts

Larisse Voufo, Andrew Lumsdaine

Center for Research in Extreme Scale Technologies (CREST)

Pervasive Technology Institute at Indiana University

C++Now'14, Aspen, CO, USA

Topics Covered

- What is ConceptClang about?
  - Open for contributions, soon.

- Ongoing theoretical progress:
  - A framework for reasoning about name binding, simply.
  - Weak hiding, a new scoping rule.
  - Structure-opening (SO) archetypes

    → Open/Extensible classes/structures for free!

- Towards a comparative study of the design space of C++ concepts

Objective

Theory: **Concepts**	Practice: **C++**
Safety	*Flexibiliy* and *Performance*
…	…
System F^G	STL
Institutions	MTL
Algebraic Specification Languages, e.g., Tecton	Numerous libraries, e.g. Boost

Objective

Practical theory: **C++ Concepts**

Flexibiliy and *Performance* with *Safety*

ConceptClang: A guide for Designing Concepts for C++

- ConceptClang implements concepts in Clang
- Concepts = Constraints-based polymorphism

In C++:	templates
In Haskell:	type classes (since Haskell 2010)
In ML:	signatures

Problem Statement

- ?
- ?
- ?
- ?
- ?
- ?
- ?
- ?
- ?

Example 1: Error Detection and Diagnosis with C++ Templates

```
vector<void*> v;
sort(v.begin(), v.end(), boost::bind(less<int>(),_1,_2));
```

```
$ clang++ test.cpp -o example
/usr/local/include/boost/bind/bind.hpp:303:16: error: no                nction for call to object
of type 'std::less<int>'
        return unwrapper<F>::unwrap(f, 0)(a[base                        pe::a2_]);
               ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
/usr/local/include/boost/bind/bind_templat                              iation of function
template specialization
      'boost::_bi::list2<boost::arg                                    r()<bool, std::less<int>,
boost::_bi::list2<void *&, void
        BOOST_BIND_RETURN l                                           0);

/usr/include/c++/4.2                                 e: in instantiation of function template
specialization
      'boost::                                cified, std::less<int>,
boost::_bi                                   arg<2> > >::operator()<void *, void *>' requested here
        i

/usr/includ                   tl_algo.h:2591:7: note: in instantiation of function template
...
2 errors ge                 .
```

Incompatible Binary Operator!

Example 2: Error Detection and Diagnosis with C++ Templates

```
vector<int> v;
sort(v.begin(), v.end(), not_equal_to<int>());
```

```
$ clang++ test.cpp -o example
$
```

Not Valid Ordering!

```
!?)
```

Example 1: Error Detection and Diagnosis with Constrained Templates

```
vector<void*> v;
constrained_sort(v.begin(), v.end(), boost::bind(less<int>(),_1,_2));
```

```
$ clang++ test.cpp -o example
test.cpp:260:2: error: no matching function for call to 'constrained_sort'
        constrained_sort(v.begin(), v.end(), boost::bind(less<int>(), _1, _2));
        ^~~~~~~~~~~~~~~
./constrained_algo.h:39:6: note: candidate template ignored: constraints check
failure [with I = __gnu_cxx::__normal_iterator<void **, std::vector<void *,
std::allocator<void *> > >, Cmp = boost::_bi::bind_t<boost::_bi::unspecified,
std::less<int>, boost::_bi::list2<boost::arg<1>, boost::arg<2> > >]
void constrained_sort(I first, I last, Cmp bin_op) {
     ^
./constrained_algo.h:38:17: note: Concept map requirement not met.
           Assignable<RandomAccessIterator<I>::value_type, ...
                     ^
./constrained_algo.h:37:3: note: Constraints Check Failed: constrained_sort.
  requires(RandomAccessIterator<I>, StrictWeakOrdering<Cmp>,
  ^
1 error generated.
```

Example 2: Error Detection and Diagnosis with Constrained Templates

```
vector<int> v;
constrained_sort(v.begin(), v.end(), not_equal_to<int>());
```

```
$ clang++ test.cpp -o example
test.cpp:261:2: error: no matching function for call to 'constrained_sort'
        constrained_sort(v.begin(), v.end(), not_equal_to<int>());
        ^~~~~~~~~~~~~~~~
./constrained_algo.h:39:6: note: candidate template ignored: constraints check
failure [with I = __gnu_cxx::__normal_iterator<int *, std::vector<int,
std::allocator<int> > >, Cmp = std::not_equal_to<int>]
void constrained_sort(I first, I last, Cmp bin_op) {
     ^
./constrained_algo.h:37:55: note: Concept map requirement not met.
  requires(RandomAccessIterator<I>, StrictWeakOrdering<Cmp>,
                                                      ^
./constrained_algo.h:37:3: note: Constraints Check Failed: constrained_sort.
  requires(RandomAccessIterator<I>, StrictWeakOrdering<Cmp>,
  ^
1 error generated.
```

Problem Statement

- Semantic errors are not detected.
- Error messages are too long and not understandable.
- Library code leaks to users space.
- **Need separate type checking**!

- Library- and preprocessor- based idioms are not enough.
  - BCCL, archetypes, enable-if, etc…
  - Context-free source-to-source transformation tools.

- **Need language support for concepts!**
- **=> More expressive power!**

Concepts for C++

- Concepts were first introduced around 1993.
- Concepts drove the design of the STL, in 1998.
- C++ libraries are designed with concepts in mind.
  - e.g. Boost, STL, …
  - Documentations in terms of concepts.

Concepts for C++

- Several conflicting designs for language support:
  - "Implicit" -- Texas A&M -- 2003
    - Simplicity and backward compatibility
  - "Explicit" -- Indiana University -- 2005
    - More expressive and generic power
  - "Pre-Frankfurt" -- Consensus… almost -- 2009
    - "Untried, too risky, not ready." – Bjarne Strousstrup
    - **Only one prototype, limited: ConceptGCC.**
  - "Palo Alto" -- A different approach -- 2011
    - A library-based perspective.
    - What are concepts? How should they be used?
    - Guide with a subset of the STL.

Concepts for C++

- Several conflicting designs for language support:
  - “Implicit” -- Texas A&M -- 2003
    - Simplicity and backward compatibility
  - “Explicit” -- Indiana University -- 2005
    - More expressive and generic power
  - “Pre-Frankfurt” -- Consensus… almost -- 2009
    - “Untried, too risky, not ready.” – Bjarne Strousstrup
    - **Only one prototype, limited: ConceptGCC.**
  - “Palo Alto” -- A different approach -- 2011
    - “Concepts-Light” -- Step 0 towards “PaloAlto” -- 2012

Concepts for C++

- Several conflicting designs for language support:
  - "Implicit" -- Texas A&M -- 2003
  - "Explicit" -- Indiana University -- 2005
  - "Pre-Frankfurt" -- Consensus… almost -- 2009
    - **Only one prototype, limited: ConceptGCC.**
  - "Palo Alto" -- A different approach -- 2011
    - "Concepts-Light" -- Step 0 towards "PaloAlto" -- 2012
- No consensus has been reached.
- Discussions are more analytical and less concrete.
- **Need a concrete basis for experiments**

==> ConceptClang.

ConceptClang

- **Picks up where ConceptGCC left off. -- 2010**
  - "Implicit" -- Texas A&M -- 2003
  - "Explicit" -- Indiana University -- 2005
  - "Pre-Frankfurt" -- Consensus… almost -- 2009
    - **Only one prototype, limited: ConceptGCC.**
  - "Palo Alto" -- A different approach -- 2011
    - "Concepts-Light" -- Step 0 towards "PaloAlto" -- 2012

ConceptClang

- **Picks up where ConceptGCC left off. -- 2010**
  - “Implicit” -- Texas A&M -- 2003
  - “Explicit” -- Indiana University -- 2005
  - “Pre-Frankfurt” -- Consensus… almost -- 2009
    - **Only one prototype, limited: ConceptGCC.**
- **First prototype -- 2011**
  - “Palo Alto” -- A different approach -- 2011
    - “Concepts-Light” -- Step 0 towards “PaloAlto” -- 2012
- **The current state is more generic.**

ConceptClang

- Treats components of concepts as first-class entities.
- Implements concepts generically, independently of design details.
- Allows to experiment with different designs.
- Primary designs of interest:
  - “Pre-Frankfurt”
  - “Palo Alto”
    - “Concepts-Light” is a variant with special properties.
- Different variants of each design are supported,
  - enabled through different compiler flags.

Outline

① Problem Statement

a. Error detection and diagnosis

b. Designing concepts for C++: A historical outline

② Concepts: Definition

a. From algorithms to concepts

b. The components

③ ConceptClang: Implementation Structure

④ Theoretical Contributions

a. Name binding framework

b. Weak hiding, a new scoping rule

c. Structure opening archetypes, or extensible structures for free

⑤ A comparative study of the design space of C++ concepts

Concepts: Definition

- Concepts are a feature for generic programming, which:
  - allows constraints-based polymorphism.
- In C++, concepts are used to constrain templates.

```
template<typename I, typename T, template Op>


T accumulate(I first, I last, T init, Op bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Concepts: Definition

- Concepts are a feature for generic programming, which:
  - allows to express algorithms and data structures in terms of properties on types, rather than types.
  - expresses and groups the properties.

```
concept InputIterator<typename X> : Iterator<X>, EqualityComparable<X> {
    ObjectType value_type = typename X::value_type;
    MoveConstructible pointer = typename X::pointer;
    SignedIntegralLike difference_type = typename X::difference_type;

    ...

    pointer operator->(const X&);
};
```

Concepts: Definition

- Concepts are a feature for generic programming, which:
  - allows to express algorithms and data structures in terms of properties on types, rather than types.
  - expresses and groups the properties.
  - preserves the efficiency of concrete implementations.

```
int sum(int* array, int n) {
  int s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}
```

```
sum(arr,3)
```

```
template<InputIterator I,…>
  requires(…)
T accumulate(I first, I last,
             T init, Op bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

```
accumulate(arr, arr+3, 0, 1)
```

Same Complexity.

Definition

- Concepts are a feature for generic programming, which:
  - allows to express algorithms and data structures in terms of properties on types, rather than types.
  - expresses and groups the properties.
  - preserves the efficiency of concrete implementations.
  - provides specialized implementations for completeness and efficiency.

```
template<ForwardIterator I,…> requires(…)
void rotate(I first, I middle, I last) { … }

template<RandomAccessIterator I,…> requires(…)
void rotate(I first, I middle, I last) { … }
```

Concepts: Definition

- Concepts are a feature for generic programming, which:
  - allows to express algorithms and data structures in terms of properties on types, rather than types.
  - expresses and groups the properties.
  - preserves the efficiency of concrete implementations.
  - provides specialized implementations for completeness and efficiency.
  - preserves or improves safety,
    - i.e. promotes separate type checking,
    - which leads to improved error detection and diagnosis.

Components of concepts

✧ Concept Definition:

- Name + parameters
- Requirements
  - Refinements

✧ Concept Model (Template):

- Concept id: name + arguments
- Requirement satisfactions
  - Refinement satisfactions

✧ Constrained Template Definition:

- Constraints specification

✧ Constrained Template Use:

- Constraints satisfaction

ConceptClang Infrastructure

- ✧ ConceptDecl:
  - is a TemplateDecl
  - is a DeclContext
  - …
  - Decl
    - Container of ConceptModelArchetype
- ✧ *TemplateDecl:
  - Container of ConceptModelArchetypes

- ✧ ConceptModelDecl:
  - is a TemplateDecl
  - is a DeclContext
  - …
  - Decl
    - Container of ConceptModelDecls
- ✧ *TemplateSpecDecl:
  - Container of ConceptModelDecls

Models vs. Archetypes

Model Archetypes:

- represent *specified* constraints.
- hold "*substituted*" requirements.
- serve as placeholders.
- refine model archetypes.
- cannot be templates.

Models:

- represent *satisfied* constraints.
- hold "*satisfied*" requirements.
- may be concrete.
- refine model.
- may refine model archetypes, if model templates.

ConceptClang Infrastructure

- ConceptDecl:
  - is a TemplateDecl
  - is a DeclContext
  - …
  - **Decl**
    - Container of ConceptModelArchetype
- *TemplateDecl:
  - Container of ConceptModelArchetypes

- ConceptModelDecl:
  - is a TemplateDecl
  - is a DeclContext
  - …
  - **Decl**
    - Container of ConceptModelDecls
- *TemplateSpecDecl:
  - Container of ConceptModelDecls

ConceptClang Infrastructure

- **Implementation is parameterized by the requirements**.
  - Requirements are
    - parsed into **declarations**,
    - *satisfied* for concrete concept models, and
    - *substituted* for concept model archetypes.
  - The *satisfaction* or *substitution* of a requirement results in new declarations in concept models or concept model archetypes.
  - Concept definitions and models are **declaration contexts**.
- **Constraints satisfaction == Refinement satisfaction**.
  - Concept definitions == template definitions.
  - Concept models == template uses.

ConceptClang Instantiations

Proposed Design	**Requirements Representation**	**Modeling Mechanism**	**Requirements Satisfaction**	**Checking Body of Constrained Template Defns**
Pre-Frankfurt Design	Pseudo-signatures	**Both** Explicit and Implicit	Collect valid candidates	Name lookup in constraints
Palo Alto Design	Use-patterns, extended with type annotations	Implicit	Find a valid expression	Match expression trees against use-patterns
ConceptClang Infrastructure	**Declarations** [extend class **Decl**]	**Both Explicit and Implicit**	**Collect valid candidates**	**Based on name lookup in constraints**
Pre-Frankfurt Instantiation	• Reuse Clang's • **1 new kind**	–	–	–
Palo Alto Instantiation	• **4 new kinds** (incl. a **dummy** kind -- for parsing use-patterns)	**Implicit only** (disable explicit)	Find a valid expression, **in addition**	Match expression trees against use-patterns, **in addition**

Name Uses in Restricted Scopes

Pre-Frankfurt:

- Constraints introduce substituted declarations in the restricted scope.
- Name binding looks up declarations in restricted scopes

```
concept C<typename P> {
 void foo(P);
}
template<typename T>
 requires C<T>
   // Adds: void foo(T);
void func(T a) { foo(a);}
```

Palo Alto:

- Constraints don't have to introduce new declarations in the restricted scope.
- Name binding matches expressions against constraints.

```
concept C<typename P> =
  requires (P a) { foo(a);
};
template<typename T>
 requires C<T>
   // To Match: foo(a)
void func(T a) { foo(a); }
```

Name Uses in Restricted Scopes

Pre-Frankfurt:

- Names in constraints shadow names in outer scope.

```
concept C<typename P> {
 void foo(P);
}
template<typename T>
 requires C<T>
   // Adds: void foo(T);
void func(T a) { foo(a);}
```

Palo Alto:

- Declarations matching constraints shadow those in outer scope---*implicitly*.
- Expression validation is not yet specified.

```
concept C<typename P> =
  requires (P a) { foo(a);
};
template<typename T>
 requires C<T>
   // To Match: foo(a)
void func(T a) { foo(a); }
```

Outline

① Problem Statement

a. Error detection and diagnosis

b. Designing concepts for C++: A historical outline

② Concepts: Definition

a. From algorithms to concepts

b. The components

③ ConceptClang: Implementation Structure

④ Theoretical Contributions

a. Name binding framework

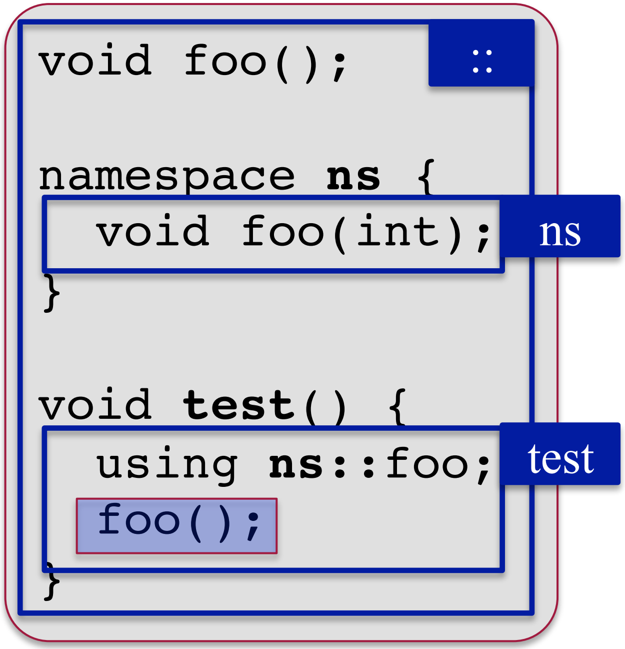
b. Weak hiding, a new scoping rule

c. Structure opening archetypes, or extensible structures for free

⑤ A comparative study of the design space of C++ concepts

Name Binding Framework

- Name binding matches *references* to *declarations*, based on *scoping rules*
  - defined by the language.
- Abstract from the fundamental notions *references*, *declarations* and *scopes*.
- Understand and specify name binding in terms of:
  - scope combinators, and
  - a **`Language`** concept.

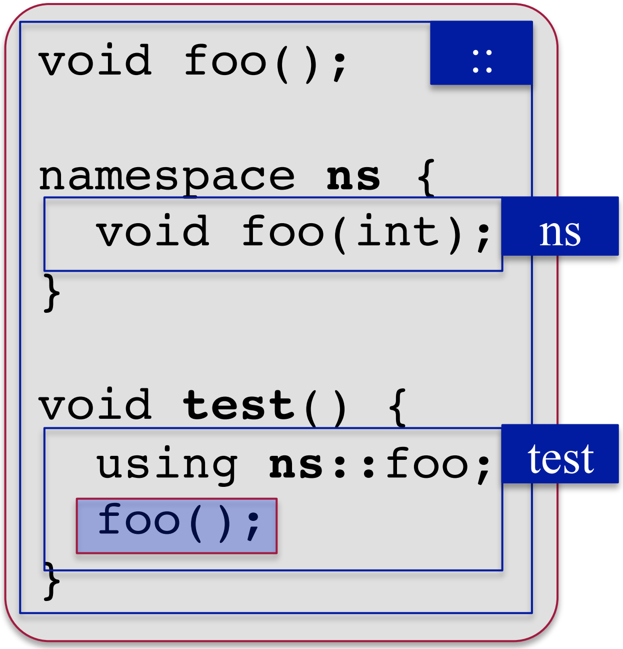


```
void foo();

namespace ns {
  void foo(int);
}

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

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

Name Binding Framework

- The scope combinators allow to express scoping rules.
- The **Language** concept encapsulates other salient properties:
  - How a declaration **matches** a reference.
  - How to **select a best viable** declaration.
  - How to interpret **ambiguity**.
- Example:
  - `foo(int)` **matches** but is **not viable**.
  - `foo()` **matches** and is **viable**.
  - Call fails.



```
void foo();

namespace ns {
  void foo(int);
}

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

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

The Combinators

- ***Hiding***: <|
  - Commonly known as "*shadowing*".
- ***Merging***: <=>
  - Usually the alternative to "*shadowing*" (e.g. module imports).
- ***Opening***: |>
  - [New name] Necessary to describe ADL.
  - A dual of *hiding*.
- ***Weak Hiding***: <=|
  - [New rule] Necessary for (C++) concepts.
  - A sweet middle between *hiding* and merging.

The Combinators

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

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

$\lhd|\ ,\ \Leftrightarrow\ ,\ |\rhd\ ,\ \Leftarrow\!|$ $\quad$ $\mathbf{Scope}_{\mathrm{Ref,Decl}} \times \mathbf{Scope}_{\mathrm{Ref,Decl}} \longrightarrow \mathbf{Scope}_{\mathrm{Ref,Decl}}$

$$\mathbf{s}_1 \Leftrightarrow \mathbf{s}_2 = \boldsymbol{lookup}_{ref}\ \mathbf{s}_1 \cup \boldsymbol{lookup}_{ref}\ \mathbf{s}_2$$

$$\mathbf{s}_1 \lhd| \ \mathbf{s}_2 = \boldsymbol{lookup}_{ref}\ \mathbf{s}_1\ ?\ \boldsymbol{lookup}_{ref}\ \mathbf{s}_1 : \boldsymbol{lookup}_{ref}\ \mathbf{s}_2$$

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

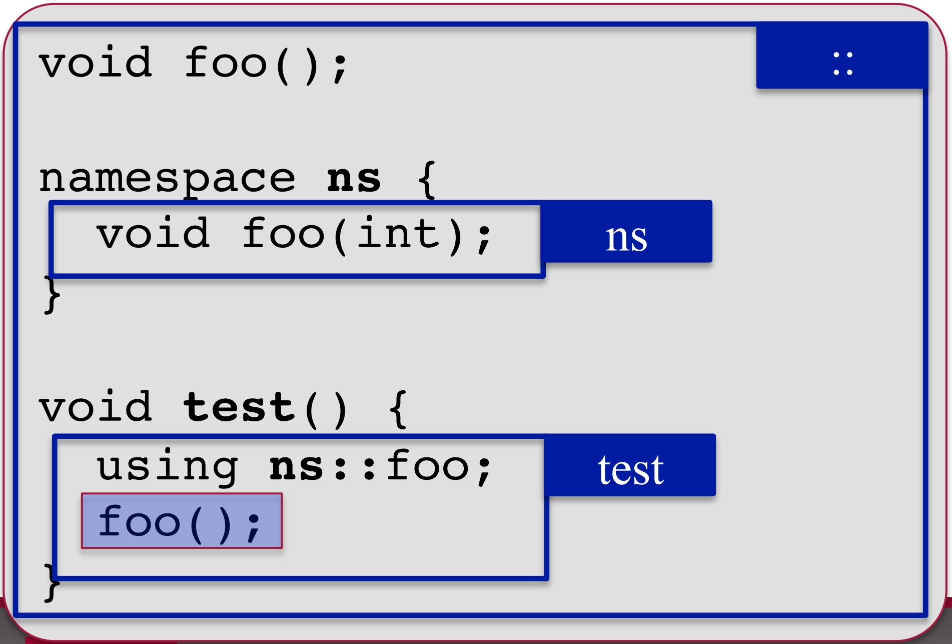
$$\mathbf{s}_1 \Leftarrow\!|\ \mathbf{s}_2 = \left(\boldsymbol{resolve}_{ref} \circ \boldsymbol{lookup}_{ref}\right) \mathbf{s}_1\ ? \\ \boldsymbol{lookup}_{ref}\ \mathbf{s}_1 : \boldsymbol{lookup}_{ref}\ \mathbf{s}_2$$

Weak Hidin

for ADL

Opening and ADL

$$\mathbf{s}_1 \Leftrightarrow \mathbf{s}_2 = \mathit{lookup}_{ref}\ \mathbf{s}_1 \cup \mathit{lookup}_{ref}\ \mathbf{s}_2$$



```
void foo();

namespace ns {
  void foo(int);
}

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

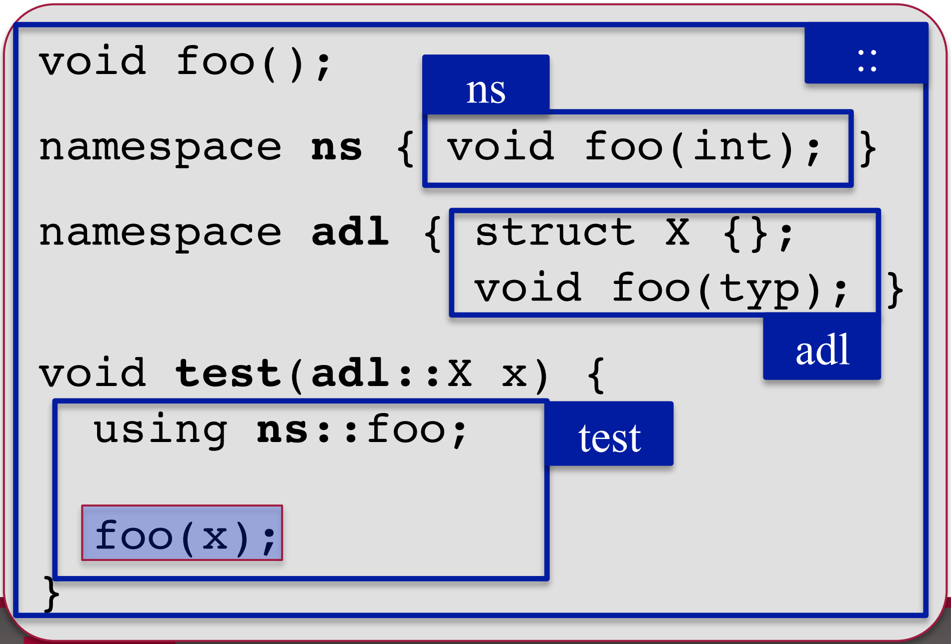
$(\mathbf{test} \Leftrightarrow \mathbf{ns}) \lhd \mathbf{::}$

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

Fails to bind **foo()**.

Opening and ADL

$$\boxed{s_1 \triangleright s_2 = \mathit{lookup}_{ref}\; s_1 \;?\; \mathit{lookup}_{ref}\; s_2 : \text{empty}}$$



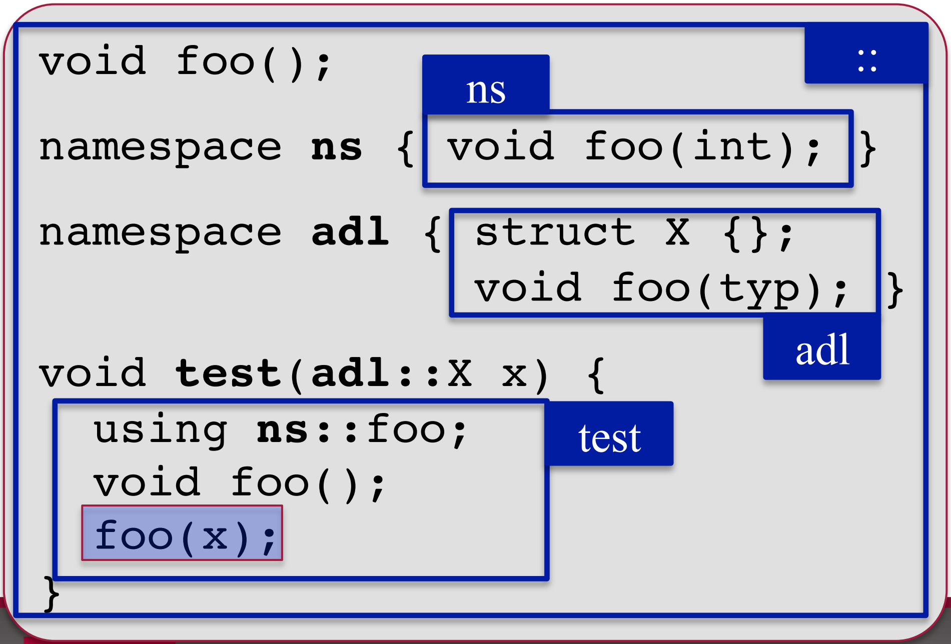
```
void foo();

namespace ns { void foo(int); }

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

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

  foo(x);
}
```

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

Finds **`ns::foo()`**;
Enables ADL;
Binds **`foo(x)`** to
`adl::foo()`.

Opening and ADL

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



```
void foo();

namespace ns { void foo(int); }

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

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

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

Finds **`ns::foo()`**;
Disables ADL;
Fails to bind **`foo(x)`**.

Generalized ADL

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

$$\begin{aligned}
\mathbf{f_{ADL}}\left(\mathbf{X}\right) &= \mathbf{X} \Leftrightarrow \mathbf{U_X} \Leftrightarrow \left(\mathbf{U_X} \rhd \left(\left(\mathbf{X} \Leftrightarrow \widetilde{\mathbf{U}}_{\mathbf{X}}\right) \lhd \mathbf{ADL}\right)\right),\\
\mathbf{S_1} \cdots \mathbf{S_s} &= \text{surrounding non-namespace scopes},\\
\mathbf{fn_{ADL}}\left(\mathbf{N}\right) &= \mathbf{N} \Leftrightarrow \mathbf{U_N} \Leftrightarrow \left(\left(\mathbf{N} \Leftrightarrow \mathbf{U_N}\right) \rhd \left(\left(\widetilde{\mathbf{N}} \Leftrightarrow \widetilde{\mathbf{U}}_{\mathbf{N}}\right) \lhd \mathbf{ADL}\right)\right),\\
\mathbf{N_1} \cdots \mathbf{N_n} &= \text{surrounding namespaces},\\
\mathbf{H} &= \text{scope where name binding is triggered from},\\
\mathbf{U_X} &= \text{using declarations in scope } \mathbf{X},\\
\widetilde{\mathbf{X}} &= \text{non-function (template) declarations in scope } \mathbf{X}, \text{ and}\\
\mathbf{ADL} &= \text{associated namespaces of reference's arguments.}
\end{aligned}$$

Generalized ADL

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

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

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

$$\begin{aligned}
\mathbf{N_1} \cdots \mathbf{N_n} &= \text{surrounding namespaces,}\\
\mathbf{H} &= \text{scope where name binding is triggered from,}\\
\mathbf{U_X} &= \text{using declarations in scope } \mathbf{X},\\
\widetilde{\mathbf{X}} &= \text{non-function (template) declarations in scope } \mathbf{X}\text{, and}\\
\mathbf{ADL} &= \text{associated namespaces of reference's arguments.}
\end{aligned}$$

CREST

Center for Research in Extreme Scale Technologies

Generalized ADL

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

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

$\mathbf{H = N_n}$

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

Other Applications

- Clarify uses of operators.
  - Builtin and member candidates take priority over outer scopes.
- Type-directed name resolution
  - does not change scoping rule.
  - changes match property of the **`Language`** concept.
- Haskell's type signatures: Documentation or Specification?
  - Depends on how name binding is interpreted.
  - Exclude or include type inference?
- Compiler integrations.
- Introduce weak hiding.

Outline

① Problem Statement

a. Error detection and diagnosis

b. Designing concepts for C++: A historical outline

② Concepts: Definition

a. From algorithms to concepts

b. The components

③ ConceptClang: Implementation Structure

④ Theoretical Contributions

a. Name binding framework

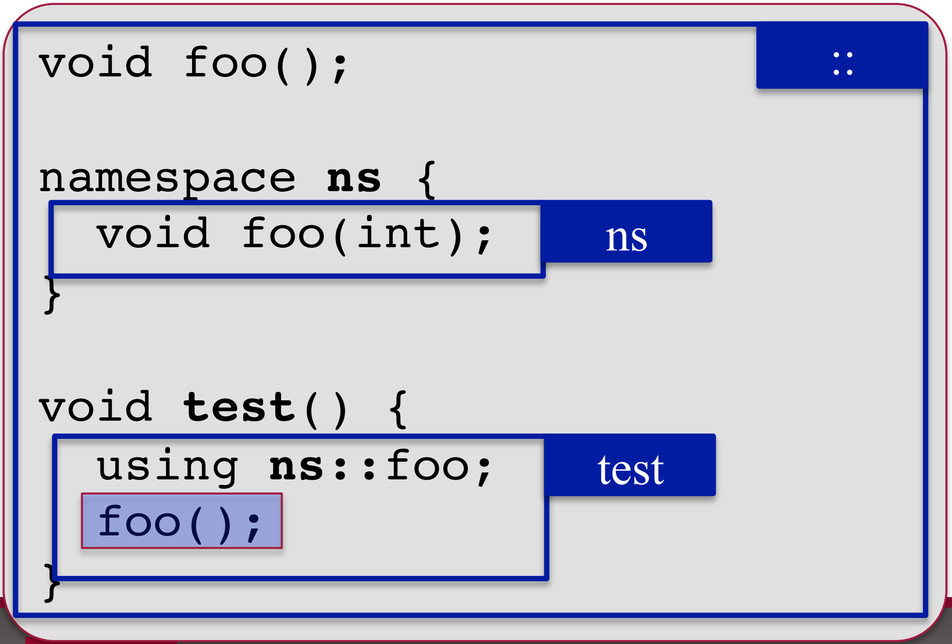
b. Weak hiding, a new scoping rule

c. Structure opening archetypes, or extensible structures for free

⑤ A comparative study of the design space of C++ concepts

Weak Hiding

$$\boxed{\mathbf{s_1} \Leftrightarrow \mathbf{s_2} = \mathit{lookup}_{ref}\ \mathbf{s_1} \cup \mathit{lookup}_{ref}\ \mathbf{s_2}}$$



```
void foo();

namespace ns {
  void foo(int);
}

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

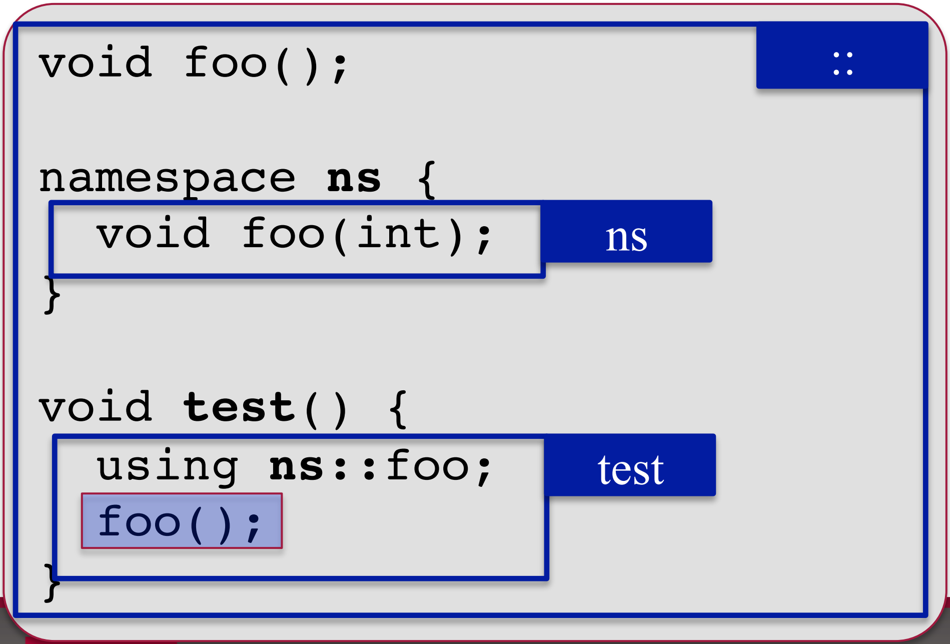
$$\boxed{(\mathbf{test} \Leftrightarrow \mathbf{ns}) \lhd \mathbf{::}}$$

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

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

Weak Hiding

$$\mathbf{s}_1 \Lleftarrow \mathbf{s}_2 = \left(\mathit{resolve}_{ref} \circ \mathit{lookup}_{ref}\right) \mathbf{s}_1 \;?\; \mathit{lookup}_{ref}\, \mathbf{s}_1 \;:\; \mathit{lookup}_{ref}\, \mathbf{s}_2$$



```
void foo();

namespace ns {
  void foo(int);
}

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

$$(\text{test} \Leftrightarrow \text{ns}) \Lleftarrow ::$$

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

Weak Hiding: Applications

- Transition from unconstrained to constrained templates
    - rejecting invalid programs,
    - preserving valid programs,
    - without changing bodies of templates.
- Implicit syntax disambiguation.
- Experiment with various properties of name binding,
    - e.g., changes in the meaning of ambiguity
    - Parameterized weak hiding
        - changes meaning of ambiguity based on scopes.
    - Bind^{x2}: A conservative implementation that
        - iterates through current mechanisms for name binding.

Weak hiding for C++ Concepts

- How to safely transition from unconstrained to constrained templates?
- The Palo Alto design has not yet specified expression validation.
  - Assume that declarations that would match the constraint are ***implicitly*** injected into the restricted scope.
- A Constraints Check Forwarding (CCF) condition
  - allows to specify a related class of invalid references.

```
concept C<Regular T>
= requires (T a)
   { foo(a); }

void foo(int);

template<typename T>
 requires C<T>
void gen_func(T a) {
  foo(a);
  foo(1);  // Pass ?
}
```

The CCF Condition

```
concept C<Semiregular T> =
    requires (T a) { foo(a); }

struct B {};
template<Semiregular T> void foo(T
void foo(B&);

// Calls to ::foo() forward responsibility for
// constraints checking from gen_func() to foo().
template<C T1, typename T2>
  requires Convertible<T2,T1> && Convertible<B,T1>
void gen_func(T1 a1, T2 a2, B b) {
    foo(a1);  // == ::foo(a1) ?
    foo(a2);  // == ::foo(a2) ?
    foo(b );  // == ::foo(b ) ?

    // Is ::foo(v) equal to or preferable to foo(v)?
}
```

Palo Alto design does not capture conversions

Merging → Subject to CCF

```
concept C<Semiregular T> =
    requires (T a) { foo(a); }

struct B {};
template<Semiregular T> void foo(T a);
void foo(B&);

// Calls to ::foo() forward responsibility for
// constraints checking from gen_func() to foo().
template<C T1, typename T2>
  requires Convertible<T2,T1> && Convertible<B,T1>
void gen_func(T1 a1, T2 a2, B b) {
    foo(a1);  // May bind to ::foo(a1).
    foo(a2);  // May bind to ::foo(a2).
    foo(b );  // May bind to ::foo(b ).

    // Is ::foo(v) equal to or preferable to foo(v)?
}
```

Practical Examples

- STL: `rotate()` and `move()`
  - Different number of parameters.
  - Uses qualified names.
- STL: `rotate()` and `operator()`
  - Palo Alto design does not cover type conversions.
- Plenoptic photography: Image rendering
  - Different parameter types.
  - Does not use qualified names.

Practical Examples

- STL: `rotate()` and `move()`
  - Is subject to CCF. Proof → Weak hiding is not needed.
  - Alternative requires extensible structures.
- STL: `rotate()` and `operator()`
  - Is subject to CCF. Proof → Weak hiding is needed.
- Plenoptic photography: Image rendering
  - Not subject to CCF.
  - Requires extensible structures for completeness.

Practical Examples

- STL: `rotate()` and `move()`
  - Is subject to CCF. Proof → Weak hiding is not needed.
  - Alternative requires **extensible structures**.
- **STL: `rotate()` and `operator()`**
  - Is subject to CCF. Proof → Weak hiding is needed.
- **Plenoptic photography: Image rendering**
  - Not subject to CCF, if using (weak) hiding.
  - Requires **extensible structures** for completeness.

STL rotate and Operators

```
// Adapted from latest release of libstdc++,
// using Palo Alto concepts.

template<RandomAccessIterator I>
requires Permutable<I>   // => move(ValueType<I>&&)
I rotate (I first, I middle, I last) { ...
  I p = first;
  ...
  if (__is_pod(ValueType<I>) ...) {
    ValueType<I> t = std::move(*p);
    std::move(p+1, p+last-first, p);
    *(p + last - first - 1) = std::move(t);

}
```

Not Ok.
No conversion from `int` to `DifferenceType<I>`

Ok.
`Last-first` has type `DifferenceType<I>`

Plenoptic Rendering

```
template<Regular PixelType>
requires IndirectlyCopyable<MultiArrayIter<PixelType>,
                            MultiArrayIter<PixelType>>
         // => move(PixelType&&)
struct Radiance {
  typedef … boost::multi_array<PixelType, 4> RadianceType;
  ...
  Radiance<PixelType> Render_Blended(...) { ...
  RadianceType Rendered(boost::e
  for ...          // for each imag
    PixelType pixel_avg; ...
    for ...          // for each
      pixel_avg += ... // integrate pixel
    ...
    Rendered[i][j][0][0] = move(pixel_avg);
    ...
  return move(new Radiance<PixelType>(move(Rendered), Ix, Iy));
}
```

Not covered by constraints, but available in outer scope

Plenoptic Rendering

```
template<Regular PixelType>
requires IndirectlyCopyab

                 // => mo
struct Radiance {
  typedef … boost::multi_array<PixelType, 4> RadianceType;
  ...
  Radiance<PixelType> Ren
  RadianceType Rendered(b
  for ...          // for ea
    PixelType pixel_avg; ...
    for ...             // for each direction
      pixel_avg += ... // integrate pixel
    ...
    Rendered[i][j][0][0] = move(pixel_avg);
    ...
  return move(new Radiance<PixelType>(move(Rendered), Ix, Iy));
}
```

May use `std::move()`.
Could access `Radiance` and `PixelType`.

Implement own `move()`?

Move with `std::move()`

- `template <class T> … std::move(T&&) … :`
  - A lot of details went into its design.
  - To use it, implement move constructor `T::T(T&&)`.
- Can we take advantage of the details in our own `move()`?

```
// MoveConstraints = constraints on std::move()
concept MoveConstructor<MoveConstraints T>
 = requires (T&& a) { T(a); }

template<class T>
requires MoveConstructor<T>
… move(T&& a) { return std::move(a); }
```

Move with `std::move()`

- A concept expresses the requirements for move constructor `T::T(T&&)`.
- A concept map implements move constructor `t::t(t&&)`,
  - for some type t.

```
// MoveConstraints = constraints on std::move()
concept MoveConstructor<MoveConstraints T>
  = requires (T&& a) { a(a); }
```

```
concept_map MoveConstructor<RT> {
  RT::RT(RT&&) { ... };
}
```

Move with `std::move()`

- A concept expresses the requirements for move constructor `T::T(T&&)`.
- A concept map implements move constructor `t::t(t&&)`,
  - for some type t.
- The instantiation of `move()` captures the concept map, and
  - transfers the implementation to `std::move()`.
  - The instantiation of `std::move()` uses the implementation.

```
RT Rendered ...
move(Rendered) // => std::move(Rendered)
               //      finds RT::RT(RT&&).
```

Outline

① Problem Statement

a. Error detection and diagnosis

b. Designing concepts for C++: A historical outline

② Concepts: Definition

a. From algorithms to concepts

b. The components

③ ConceptClang: Implementation Structure

④ Theoretical Contributions

a. Name binding framework

b. Weak hiding, a new scoping rule

c. Structure opening (SO) archetypes, or extensible structures for free

⑤ A comparative study of the design space of C++ concepts

Extensible Structures for Free

```
struct DataType {
  DataType foo() {..}
};

concept Extension<typename T> {
  int T::bar();
}

concept_map Extension<DataType> {
  int DataType::bar() {…}
}

template<Extension T>
void new_test(T e) {
  int res = e.bar();
}

int main () { new_test(DataType()); }
```

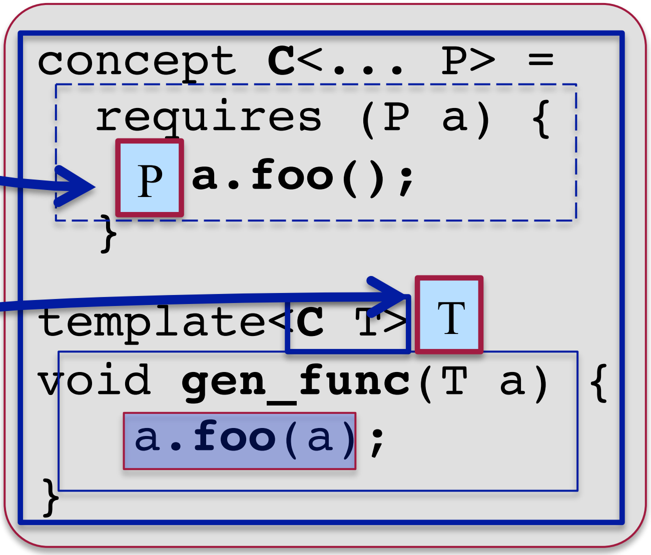
Encapsulate extensions in a concept using associated member declarations.

Provide extended function implementations via concept models.

Write all (future) generic applications using concepts

Structure Opening (SO) Archetypes

- SO archetypes are implementation artifacts (in ConceptClang)
  - that encapsulate extensions of a given structure.
- Two kinds of archetypes:
  - Elementary Archetypes,
    - in concept definitions and models.
    - Lookup in refinements too.
  - Join Archetypes,
    - in restricted scopes,
    - merge all archetypes in all constraints.
    - Lookup in all merged archetypes



```
concept C<... P> =
  requires (P a) {
   P a.foo();
  }

template<C T> T
void gen_func(T a) {
    a.foo(a);
}
```

Structure Opening (SO) Archetypes

- SO archetypes are implementation artifacts (in ConceptClang)
  - that encapsulate extensions of a given structure.
- An archetype of X is X in every way except when qualified name lookup is requested.
  - Extends Clang's `HasSame*Type()` procedures.
- Name lookup in an SO archetype follows two different paths:
  - outer() → outer restricted scopes (if weak hiding is enabled).
  - refined() → refinements.
- Name rebinding captures extended implementations at instantiation time.

Extensible Structures

- Solving the expression problem:
  - The functional way, e.g. Haskell:
    - Operations are extensible, datatypes not so much.
    - Add datatypes to operations via type classes.
  - The object-oriented way, e.g. Java, C++
    - Datatypes are extensible, operations not so much
    - Add operations to datatypes via open/extensible classes/structures.
    - e.g MultiJava, Ruby.
- Research on extensible structures:
  - New languages: OCAML, OML, ML≤, EML, MultiJava, Ruby, etc…
  - Ongoing workaround are complex:
    - Visitor pattern, double-dispatching, etc…

Outline

① Problem Statement

  a. Error detection and diagnosis
  b. Designing concepts for C++: A historical outline

② Concepts: Definition

  a. From algorithms to concepts
  b. The components

③ ConceptClang: Implementation Structure

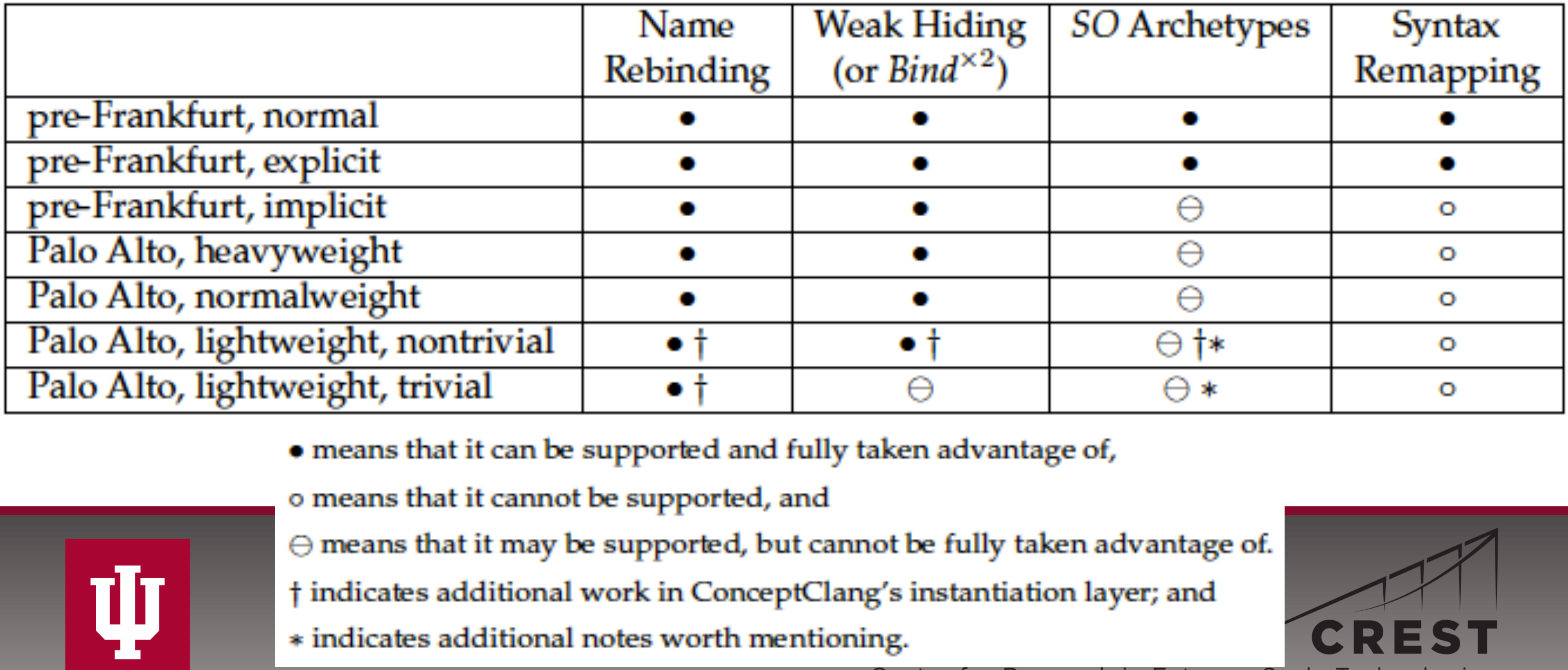
④ Theoretical Contributions

  a. Name binding framework
  b. Weak hiding, a new scoping rule
  c. Structure opening (SO) archetypes, or extensible structures for free

⑤ A comparative study of the design space of C++ concepts

A Comparative Study

- For each design variant, how much of concepts can we exploit?
  - Each feature allows to push boundaries for safety, expressiveness, and genericity.



	Name Rebinding	Weak Hiding (or *Bind*$^{\times 2}$)	*SO* Archetypes	Syntax Remapping
pre-Frankfurt, normal	●	●	●	●
pre-Frankfurt, explicit	●	●	●	●
pre-Frankfurt, implicit	●	●	⊖	○
Palo Alto, heavyweight	●	●	⊖	○
Palo Alto, normalweight	●	●	⊖	○
Palo Alto, lightweight, nontrivial	● †	● †	⊖ †∗	○
Palo Alto, lightweight, trivial	● †	⊖	⊖ ∗	○

● means that it can be supported and fully taken advantage of,

○ means that it cannot be supported, and

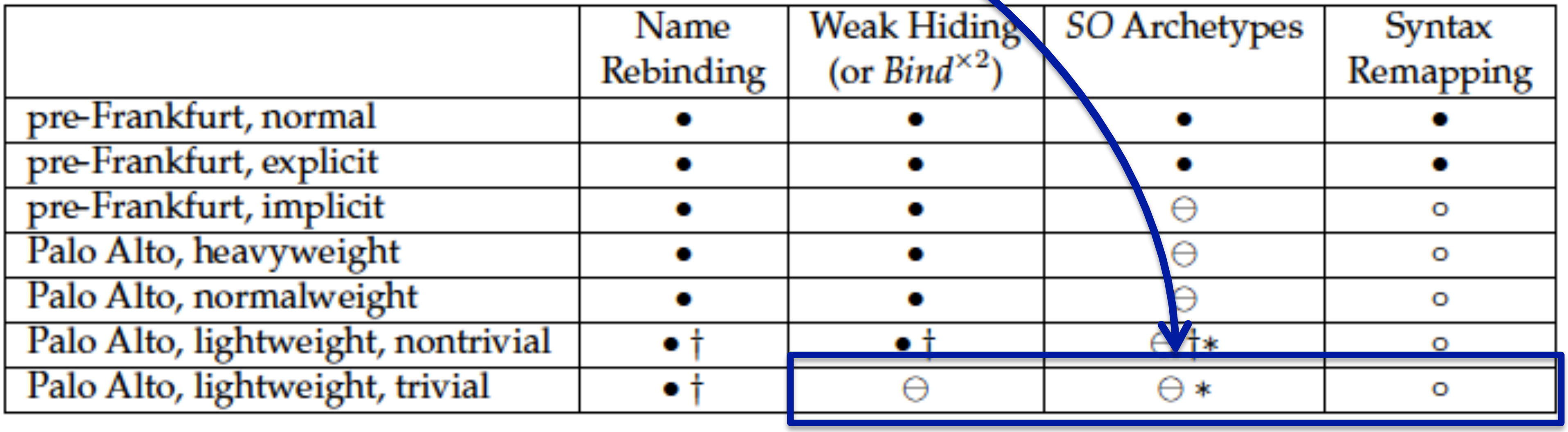
⊖ means that it may be supported, but cannot be fully taken advantage of.

† indicates additional work in ConceptClang's instantiation layer; and

∗ indicates additional notes worth mentioning.

A Comparative Study

- Concepts-Light is a trivial lightweight variant of the Palo Alto design



	Name Rebinding	Weak Hiding (or *Bind*$^{\times 2}$)	*SO* Archetypes	Syntax Remapping
pre-Frankfurt, normal	•	•	•	•
pre-Frankfurt, explicit	•	•	•	•
pre-Frankfurt, implicit	•	•	⊖	○
Palo Alto, heavyweight	•	•	⊖	○
Palo Alto, normalweight	•	•	⊖	○
Palo Alto, lightweight, nontrivial	• †	• †	⊖ † ∗	○
Palo Alto, lightweight, trivial	• †	⊖	⊖ ∗	○

Safety with Name Rebinding

```
template<typename T1, typename T2>
requires __is_valid_expr(foo(declval<T1>(),declval<T2>())) &&
         ImplicitlyConvertible<T1,T2>
void my_func(T1 a, T2 b) {
  foo(a,b); foo(a,a); // ***
}

class B {};  class A : public B { };

void foo(A,B) { };
void foo(B,A) { }; // or a templated form of foo().

int main(int argc, const char* argv[]) {
A a; B b;
foo(a,b); // is a valid expression. Binds to ’void foo(A,B)’.
          // ’void foo(B,A)’ is not a viable candidate.
my_func(a,b); // ***
//*** Constraints satisfaction succeeds, but instantiation fails ***
// because the instantiation of ’my_func’ requires ’void foo(A,A)’.
}
```

Conclusion

- Implementing concepts for C++ leads to theoretical advances in the areas of name binding and the expression problem.
- Main contributions:
  - ConceptClang: the compilation model and implementation.
  - Name binding framework.
  - Weak hiding and **Bind**^{x2}.
  - Structure-opening archetypes → Extensible structures for free.
  - A comparative study of the design space of C++ concepts.
    - Some resolvable concerns about Concepts-Light.

Outlook

- Complete ConceptClang, covering more practical examples:
  - STL, Palo Alto and pre-Frankfurt concepts, concepts from the EOP book, BGL, MTL, etc…
- Investigate the satisfaction of the CCF condition.
- Integrate our scope combinators in current compilers.
- Formalize SO archetypes, relating them to ongoing research.
- More comparative studies of the design space of concepts.

ConceptClang: Current State

- > ~30,000 lines of code.
- Essential structure has been implemented.
  - Includes weak hiding, not constraints, and SameType constraints.
  - Basic tests in all supported design variants.
    - Programs: Apple-to-Apple and Mini-BGL.
  - Few core features are in development.
    - E.g. SO archetypes, usepatterns → pseudosignatures, …
- Open for contributions in
  - testing, debugging and implementing missing components.

Thank You!

- ConceptClang:

  [http://www.crest.iu.edu/projects/conceptcpp]

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

- Name binding framework:

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