ConceptClang: Towards A Compilation Model for C++ Concepts

Andrew Lumsdaine, **Larisse Voufo**
Center for Research in Extreme Scale Technologies (CREST)
Pervasive Technology Institute at Indiana University

C++Now 2012

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

Outline

① Concepts:

1. An Introduction
   - For generic programming, in C++
   - The elementary components
2. Implementing Concepts
   - Implementation considerations
   - Design alternatives
   - Towards ConceptClang

② ConceptClang: The Compilation Model

- The components
- Essential data structures and procedures
- Preliminary observations
- Case study: Type-checking (constrained) templates
- Open questions

Objective

- Implement concepts for C++
  - in a modular and generic fashion.
- Define an abstract layer for realizing concepts designs.
  - Independent from any alternative design.
  - Extensible to any given design.
- Concretely assess the implications of concepts designs on both implementation and usability.
  - e.g. The implementation of concepts must be clearly separated from that of the current C++ standard.
- Highlight similarities and subtle differences between alternative designs
  - The implementation must clearly distinguish between an **infrastructure** layer and **instantiations** layer.

ConceptClang is…

- an implementation of C++ concepts in Clang
  - Clang is an LLVM compiler front-end for the C family of languages,
- implemented (and documented) in a modular and generic fashion,
  - isolating ConceptClang from Clang, whenever possible.
  - Isolating the infrastructure layer from design-instantiations layers.
- currently exploring the support for two main designs:
  - TR **N2914=09-0104** -- a.k.a. "**pre-Frankfurt**" design
  - TR **N3351=12-0041** -- a.k.a. "**Palo Alto**" design
- providing alternative options via compiler flags.
  - e.g. disabling implicit or explicit concepts capabilities.
- under development at Indiana University
  - Center for Research in Extreme Scale Technologies
  - https://www.crest.iu.edu/projects/conceptcpp/

ConceptClang from Clang

- Four main reasons:
  - Carefully designed coding guidelines
  - Modern C++ implementation technology
    - Highly structured code
    - Easily understandable
    - Modular—Library-based approach
  - Follows the C++ standard strictly
    - Code portability
  - License allows extension and experimentation
- Gain:
  - Conveniently reusable implementations
    - Error detection, diagnosis, and presentation
  - Reliable implications analysis

ConceptGCC, A Sister Project

- ConceptGCC is a prototype for the pre-Frankfurt design.
- ConceptGCC is implemented in GCC.
  - Basis of compilation model: concepts as class templates…
- ConceptGCC differs from ConceptClang in the basis of the compilation model.
  - ConceptClang treats concepts components as first-class entities of the C++ language.
- The development of ConceptGCC has been on hiatus since 2009.

Outline

① **Concepts:**

1. **An Introduction**
   - **For generic programming, in C++**
   - **The elementary components**
2. **Implementing Concepts**
   - **Implementation considerations**
   - **Design alternatives**
   - **Towards ConceptClang**

② ConceptClang: The Compilation Model

- The components
- Essential data structures and procedures
- Preliminary observations
- Case study: Type-checking (constrained) templates
- Open questions

Generic Programming

- Finds abstract representations of efficient software components.
  - The lifting must **preserve efficiency.**
  - The lifting and reuse must be **safe.**
- Expresses algorithms with minimal assumptions about data abstractions and vice-versa.
  - Broadly adaptable and interoperable software components.
- Provides specializations if necessary, for efficiency.
- Picks the most efficient implementation.

Generic Programming: Motivation

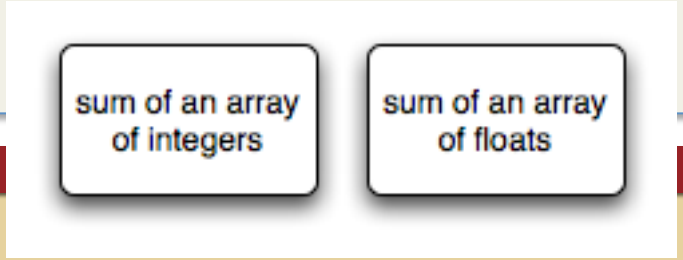
EXAMPLE: ADDING NUMBERS

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

// Array of floating-point numbers:
float sum(float* array, int n) {
  float s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}
```

Generic Programming: Motivation

EXAMPLE: ADDING NUMBERS

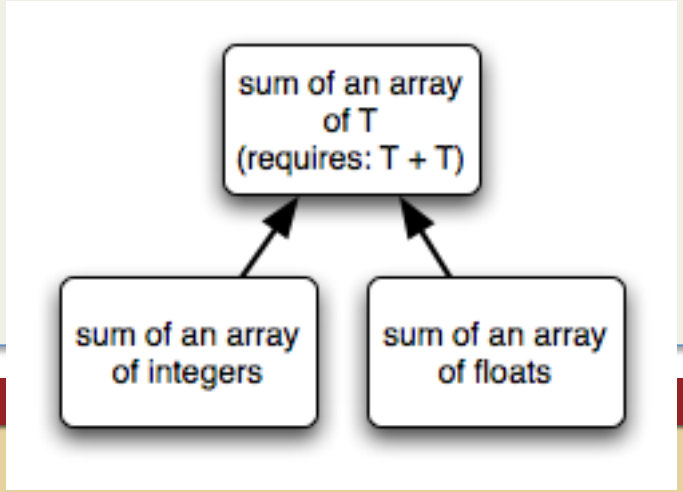
```
// Array of integers:
int sum(int* array, int n) {
  int s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}

// Array of floating-point numbers:
float sum(float* array, int n) {
  float s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}
```



Generic Programming: Motivation

EXAMPLE: LIFTING SUMMATION

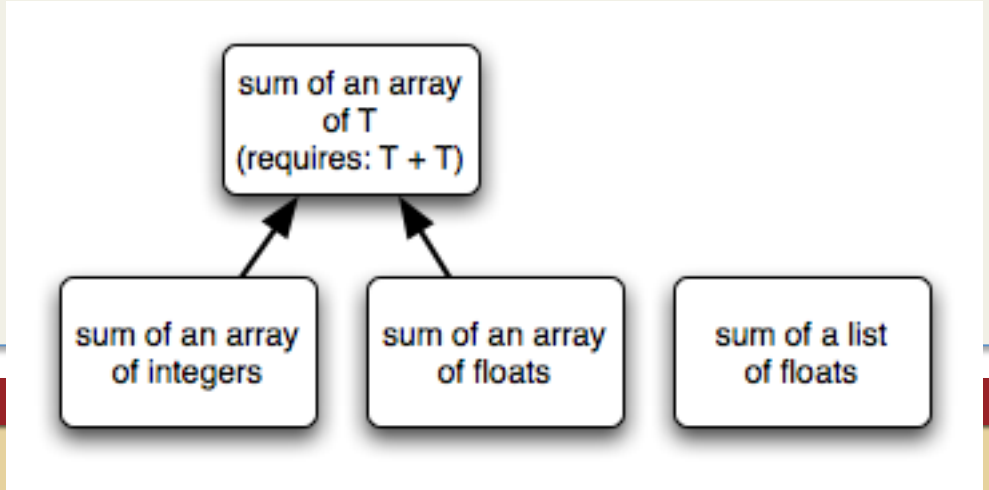
```
// Array of integers:
template<typename T>
T sum(T* array, int n) {
  T s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}
```



Generic Programming: Motivation

EXAMPLE: LIFTING SUMMATION

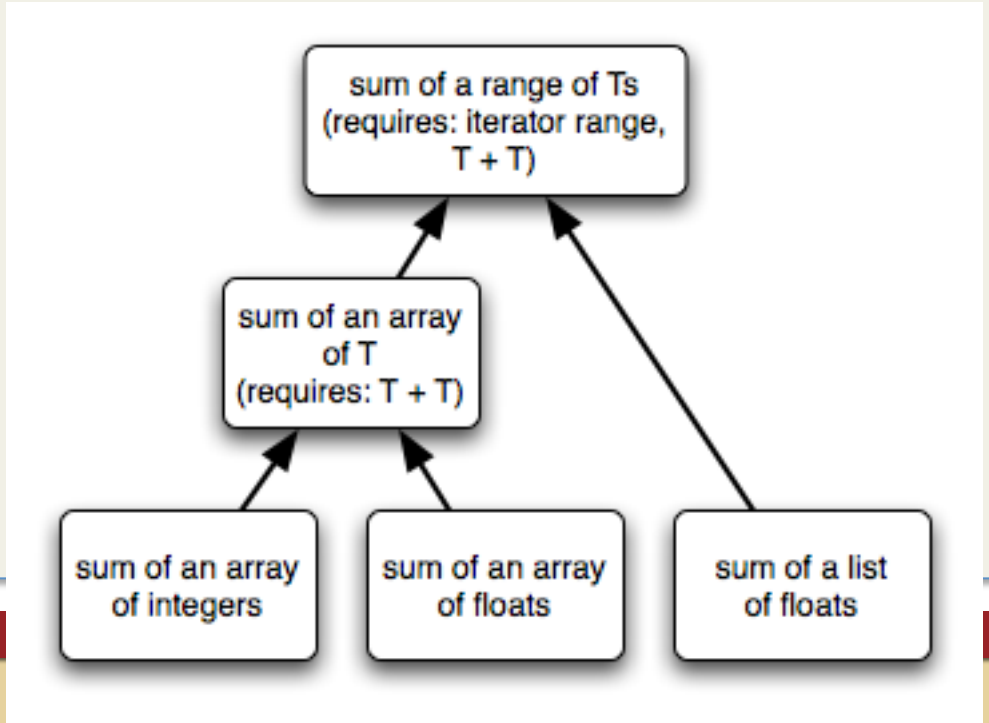
```
int sum(int* array, int n) { ... }
float sum(float* array, int n) { ... }
...
// List of floating-point numbers
double sum(list_node* first, list_node* last) {
  double s = 0;
  while (first != last) {
    s = s + first->data;        first = first->next; }
  return s;
}
...
```



Generic Programming: Motivation

EXAMPLE: LIFTING SUMMATION

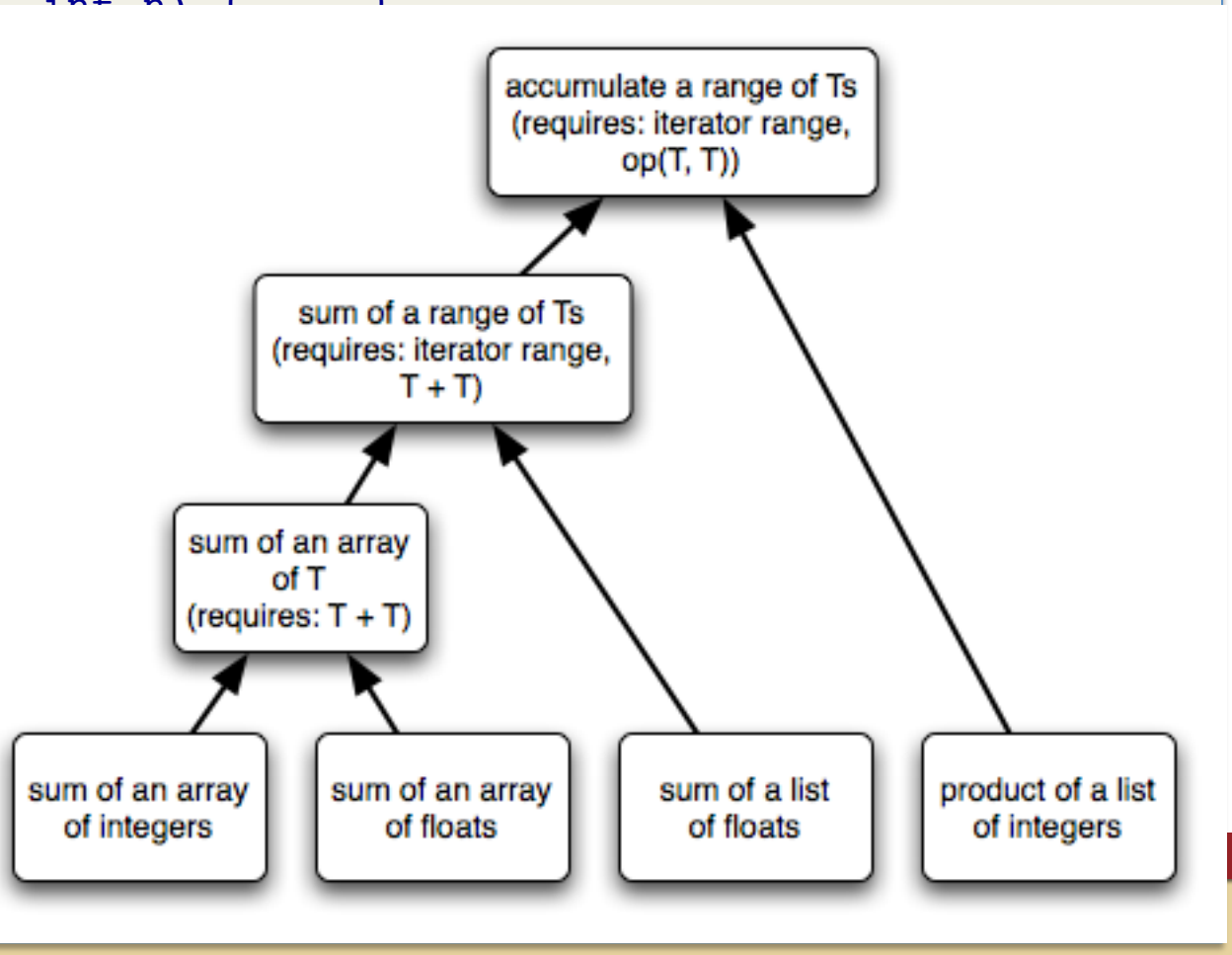
```
template<typename Iter, typename T>
T sum(Iter first, Iter last, T init) {
  for (; first != last; ++first)
    init = init + *first);
  return init;
}
```



Generic Programming: Motivation

EXAMPLE: ACCUMULATING ELEMENTS

```
int sum(int* array, int n) {...}
float sum(float*
...
double sum(list_n
  double s = 0;
  while (first !=
    s = s + first
  return s;
}
...
int prod(list_nod
  int s = 1;
  while (first !=
    s = s * first
  return s;
}
...
```



Generic Programming: Motivation

EXAMPLE: ACCUMULATE

- Define a generic function:

```
template<typename Iter, typename T, typename BinOp>
T accumulate(Iter first, Iter last, T init, BinOp bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

- Simply reuse the generic function for varying purposes:

```
// Adding integers stored in vectors:
vector<int> v;
int i = accumulate(v.begin(), v.end(), 0, plus<int>());

// Multiplying floating-point numbers stored in sets:
set<double> s;
double i = accumulate(s.begin(), s.end(), 0, multiplies<double>());

...
```

Templates: Compiler Mechanism

PARSING

Template Definition:

```
template<typename Iter,
         typename T,
         typename BinOp>
T accumulate(…) { … }
```

Template Use:

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check

INSTANTIATION

Code Generation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op)
{ ... }
```

Once!

Specialization:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

Generic Programming

- Different languages support it in various capacities:

  Genericity by …

  - Value: e.g. function abstraction
  - Type: e.g. (parametric or adhoc) polymorphism
  - Function: e.g. functions as values
  - Structure: e.g. requirements and operations on types
  - Property: e.g. properties on type
  - Stage: e.g. meta-programming
  - Shape: e.g. datatype-generic

  cf. **"Datatype Generic Programming"**. Gibbons. [In Spring School on Datatype-Generic Programming, volume 4719 of Lecture Notes in Computer Science. Springer-Verlag.]

- C++ supports generic programming via **templates**.

Concepts: Motivation

EXAMPLE: ACCUMULATE

- Define a generic function:

```
template<typename Iter, typename T, typename BinOp>
T accumulate(Iter first, Iter last, T init, BinOp bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

- Requirements:
  - Used operators are defined: `!=, ++, *, =`
  - The uses of operators are valid
  - Some operations are amortized constant time: `!=, ++, *`
  - Input parameter values can be copied and copy-constructed

- **Concepts** express and group the above requirements.
  - Types must satisfy the requirements.

Concepts: Motivation

EXAMPLE: ACCUMULATE

- Define a generic function:

```
template<typename Iter, typename T, typename BinOp>
T accumulate(Iter first, Iter last, T init, BinOp bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

- Requirements:
  - **`Iter`** satisfies the requirements of an **input iterator**
    e.g. `!=, ++, *`
  - **`BinOp`** satisfies the requirements of a **binary function**.
  - The **result of `bin_op()`** is **convertible to** the type of the elements.

- In the STL: Input Iterator, Binary Function, Assignable.

Concepts: Definition

- Concepts are an abstraction mechanism:
  - Group requirements that generic components impose on their types:
    - Syntactic properties: e.g. types, name declarations, valid expressions
    - Semantic properties: e.g. axioms
    - Complexity guarantees
  - Express algorithms in terms of the properties on their types.
- Concepts are an essential component of generic programming:
  - Constraints on types, or type predicates.
  - Aim: **Safe and efficiency-preserving reusability** of software components.
    - **Efficiency-preserving**:

      Instantiated generic component vs. initial concrete algorithm
    - **Safety**:

      Guaranteed instantiation → Separate type checking

Concepts in Programming Languages

	C++	SML	OCaml	Haskell	Eiffel	Java	C#	Cecil
Multi-type concepts	-	●	○	●*	○	○	○	◒
Multiple constraints	-	◒	◒	●	○†	●	●	●
Associated type access	●	●	◒	●*	◒	◒	◒	◒
Constraints on assoc. types	-	●	●	●	◒	◒	◒	●
Retroactive modeling	-	●	●	●	○	○	◒	●
Type aliases	●	●	●	●	○	○	○	○
Separate compilation	○	●	◒	●	●	●	●	◒
Implicit arg. deduction	●	○	●	●	○	●	◒	◒

*Using the multi-parameter type class extension to Haskell (Peyton Jones *et al.*, 1997).
*Using the functional dependencies extension to Haskell (Jones, 2000).
†Planned language additions.

Table 1: *The level of support for important properties for generic programming in the evaluated languages. A black circle indicates full support, a white circle indicates poor support, and a half-filled circle indicates partial support. The rating of "-" in the C++ column indicates that C++ does not explicitly support the feature, but one can still program as if the feature were supported due to the permissiveness of C++ templates.*

- "**An extended Comparative Study of Language Support for Generic Programming**". Garcia et. Al. [J. Funct. Program., 17(2):145–205, Mar. 2007].

Concepts in C++

- Concepts are supported as library feature:
  - Standard Template Library: Designed based on concepts
    - Defines concepts in documentation
  - Suggestive names: Express some properties on types
    - Not checked by the compiler
    - Cannot express complex requirements

SUGGESTIVE NAMES EXAMPLE: ACCUMULATE

```
// Assignable ?
template<typename InputIterator, typename T, typename BinaryFunction>
T accumulate(InputIterator first, InputIterator last,
             T init, BinaryFunction bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Concepts in C++

- Concepts are supported as library feature:
  - Standard Template Library: Designed based on concepts
  - Suggestive names: Express some properties on types
  - Type-traits, tag dispatching, etc…: Express properties on types
  - Boost Concept Checking Library: Emulates concepts-based approaches
- Library support: Issues with usability
- Library support: insufficient

Concepts in C++

- Concepts are supported as a library feature:
  - Standard Template Library: Designed based on concepts
  - Suggestive names: Express some properties on types
  - Type-traits, tag dispatching, etc…: Express properties on types
  - Boost Concept Checking Library: Emulates concepts-based approaches
- Library support: Issues with usability
  - Suggestive names: not expressive enough
  - Other methods: too verbose and complex
  - A language for experts ?
- Library support: Insufficient
  - Concepts are not checked by the compiler
  - **Templates are not separately type-checked**

    ➔ Faulty error detection and diagnosis

Problem: Error detection and diagnosis

- Late detection
  - During instantiation
- Complex and lengthy diagnoses
  - Encapsulation breaks
  - Library internals leak to users space
- Worse: Non-detected semantic errors

Templates: Problem – Example 0

- Template Use:

```
vector<void*> v;
int i = accumulate(v.begin(), v.end(), 0, plus<int>());
```

- Compiler error:

```
$ clang++ test.cpp -o example
/usr/include/c++/4.2.1/bits/stl_numeric.h:115:11: error: no matching function for
call to object of type 'std::plus<int>'
        __init = __binary_op(__init, *__first);
                 ^~~~~~~~~~~
test.cpp:221:10: note: in instantiation of function template specialization
      'std::accumulate<__gnu_cxx::__normal_iterator<void **, std::vector<void *,
std::allocator<void *> > >, int, std::plus<int> >' requested here
        int i = accumulate(v.begin(), v.end(), 0, plus<int>());
                ^
/usr/include/c++/4.2.1/bits/stl_function.h:137:7: note: candidate function not
viable: cannot convert argument of incomplete type 'void *' to 'const int'
      operator()(const _Tp& __x, const _Tp& __y) const
      ^
1 error generated.
```

Templates: Problem – Example 0.5

- Template Use:

```
vector<void*> v;
int i = accumulate(v.begin(), v.end(), 0,
                   boost::bind(plus<int>(),_1,_2));
```

- Compiler error:

```
$ clang++ test.cpp -o example
/usr/local/include/boost/bind/bind.hpp:303:16: error: no matching function for call to object
of type 'std::plus<int>'
        return unwrapper<F>::unwrap(f, 0)(a[base_type::a1_], a[base_type::a2_]);
               ^~~~~~~~~~~~~~~~~~~~~~~~~
/usr/local/include/boost/bind/bind_template.hpp:61:27: note: in instantiation of function
template specialization
      'boost::_bi::list2<boost::arg<1>, boost::arg<2> >::operator()<int, std::plus<int>,
boost::_bi::list2<int &, void *&> >' requested here
        BOOST_BIND_RETURN l_(type<result_type>(), f_, a, 0);
                          ^
/usr/include/c++/4.2.1/bits/stl_numeric.h:115:11: note: in instantiation of function template
specialization
      'boost::_bi::bind_t<boost::_bi::unspecified, std::plus<int>,
boost::_bi::list2<boost::arg<1>, boost::arg<2> > >::operator()<int, void *>' requested here
        __init = __binary_op(__init, *__first);
                 ^
test.cpp:222:10: note: in instantiation of function template specialization
...
1 error generated.
```

Templates: Problem – Example 1

- Template Use:

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

- Compiler error:

```
$ clang++ test.cpp -o example
/usr/local/include/boost/bind/bind.hpp:303:16: error: n                ction for call to object
of type 'std::less<int>'
        return unwrapper<F>::unwrap(f, 0)(a[base                    e::a2_]);
               ^~~~~~~~~~~~~~~~~~~~~~~~~~
/usr/local/include/boost/bind/bind_templa                    antiation of function
template specialization
      'boost::_bi::list2<boost::ar                    rator()<bool, std::less<int>,
boost::_bi::list2<void *&, voi
        BOOST_BIND_RETURN                    a, 0);

/usr/include/c++/4.2                    note: in instantiation of function template
specialization
      'boost                    nspecified, std::less<int>,
boost::_b                    st::arg<2> > >::operator()<void *, void *>' requested here
        if                    t))

/usr/include                    ts/stl_algo.h:2591:7: note: in instantiation of function template
...
2 errors gen                    ced.
```

Incompatible Binary Operator!

Templates: Problem – Example 2

- Template Use:

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

- Compiler error:

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

Not Valid Ordering!

Templates: Solution – Example 1

- Template Use:

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

- Compiler error:

```
$ 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.
```

Templates: Solution – Example 2

- Template Use:

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

- Compiler error:

```
$ 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.
```

Concepts for C++

- A language support for concepts would:
  - Improve the usability and safety of templates
  - Simplify existing practices
    - More expressive
    - Low-barrier to entry
  - Improve error detection and diagnosis
    - At the interface between library authors and users space.
- A language support for concepts requires:
  - New syntactic constructs
    - To fully express the requirements on algorithms
  - Extended compiler mechanism
    - Check the requirements on algorithms == **Constraints satisfaction**.

Constrained Templates

- Extend template definitions:
  - with a **`requires`** clause: == **Constraints specification**.

```
template<typename I, typename T, typename Op>
  requires(InputIterator<I>, BinaryFunction<Op>,
           Assignable<InputIterator<I>::value_type,
                      BinaryFunction<Op>::result_type>)
T accumulate(I first, I last, T init, Op bin_op);
```

  - Using a simplified form:
    - For more complex constraints: use a **requires** clause

```
template<InputIterator I, typename T, BinaryFunction Op>
  requires(Assignable<InputIterator<I>::value_type,
                      BinaryFunction<Op>::result_type>)
T accumulate(I first, I last, T init, Op bin_op);
```

Concepts: Elementary Components

✧ Concept Definition:

- Name + parameters
- Requirements

✧ Constrained Template Definition:

- Constraints specification

EXAMPLE (SGI STL DOCUMENTATION)

```
Concept: Input Iterator [ X ]
...
Associated types
    value type [ T ]
    distance type
Valid expressions          Return type
    ++i                    X&
    (void)i++
    *i++                   T
Expression semantics
    ...
    (void)i++   equiv. to (void)++i
    ...
Complexity guarantees
...
```

Concepts: Elementary Components

✧ Concept Definition:

- Name + parameters
- Requirements

✧ Modeling Mechanism:

- Matches types to concepts
- Specifies: What and How.

EXAMPLE (SGI STL DOCUMENTATION)

```
Concept: Input Iterator [ X ]
...
Associated types
    value type [ T ]
    distance type
Valid expressions          Return type
    ++i                    X&
    (void)i++
    *i++                   T
Expression semantics
    ...
    (void)i++   equiv. to (void)++i
    ...
Complexity guarantees
...
```

✧ Constrained Template Use:

- Constraints satisfaction

Concepts: Elementary Components

✧ Concept Definition:

- Name + parameters
- Requirements

✧ Concept Model:

- Concept id: name + arguments
- Requirement satisfactions

EXAMPLE (SGI STL DOCUMENTATION)

```
Concept: Input Iterator [ X ]
...
Associated types
    value type [ T ]
    distance type
Valid expressions          Return type
    ++i                    X&
    (void)i++
    *i++                   T
Expression semantics
    ...
    (void)i++   equiv. to (void)++i
    ...
Complexity guarantees
...
```

EXAMPLE

```
int* is a model of Input Iterator
...
Associated type values
    value type      = int
    distance type   = ptrdiff_t
Valid expressions          Return type
    ++i                    int&
    (void)i++
    *i++                   int
Expression semantics
    ...
    (void)i++   equiv. to (void)++i
    ...
Complexity guarantees
...
```

Concept Definitions and Models

EXAMPLE (PRE-FRANKFURT)

```
concept_map InputIterator<vector<int>::iterator> {

  // Associated type satisfactions – implicit in this case
  //typedef value_type vector<int>::iterator::value_type;
  // . . .

  // Associated function satisfactions
  pointer operator->(const vector<int>::iterator&) { … }
};
```

- References:
  - C++ Standards Committee. *Working Draft, Standard for Programming Language C++. Technical Report N2914=09-0104*, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, June 2009.

Concepts: Elementary Components

✧ Concept Definition:

- Name + parameters
- Requirements
- Refinements
  - Extends requirements

✧ Constrained Template Definition:

- Constraints specification

EXAMPLE (SGI STL DOCUMENTATION)

```
Concept: Input Iterator [ X ]
Refinement of:
    Trivial Iterator
Associated types
    value type [ T ]
    distance type
Valid expressions          Return type
    ++i                    X&
    (void)i++
    *i++                   T
Expression semantics
    ...
    (void)i++   equiv. to (void)++i
    ...
Complexity guarantees
...
```

Concepts: Elementary Components

✧ Concept Definition:

- Name + parameters
- Requirements
- Refinements
  - Extends requirements

✧ Concept Model:

- Concept id: name + arguments
- Requirement satisfactions
- Refinement satisfactions
  - One for each refinement

EXAMPLE (SGI STL DOCUMENTATION)

```
Concept: Input Iterator [ X ]
Refinement of:
    Trivial Iterator
...
```

EXAMPLE

```
int* is a model of Input Iterator
int* is also a model of
    Trivial Iterator
...
```

Concepts: Elementary Components

✧ Concept Definition:

- Name + parameters
- Requirements
- Refinements
  - Extends requirements

✧ Concept Model (Template):

- Concept id: name + arguments
- Requirement satisfactions
- Refinement satisfactions
  - One for each refinement

EXAMPLE (SGI STL DOCUMENTATION)

```
Concept: Input Iterator [ X ]
Refinement of:
    Trivial Iterator
...
```

EXAMPLE

```
my_type* is a model of Input
Iterator
my_type* is also a model of
    Trivial Iterator
...
```

Concept Definitions and Models

EXAMPLE (PRE-FRANKFURT)

```
template<typename T>
concept_map InputIterator<vector<T>::iterator> {

  // Associated type satisfactions – implicit in this case
  //typedef value_type vector<T>::iterator::value_type;
  // . . .

  // Associated function satisfactions
  pointer operator->(const vector<T>::iterator&) { … }
};

// Refinements and associated requirements are
// satisfied by defining models for them.
```

- References:
  - C++ Standards Committee. *Working Draft, Standard for Programming Language C++. Technical Report N2914=09-0104*, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, June 2009.

Concepts: Elementary Components

✧ Concept Definition:

- Name + parameters
- Requirements
- Refinements
  - Extends requirements

✧ Concept Model (Template):

- Concept id: name + arguments
- Requirement satisfactions
- Refinement satisfactions
  - One for each refinement

✧ Constrained Template Definition:

- Constraints specification

✧ Constrained Template Use:

- Constraints satisfaction

Concepts: Recap

- Concepts group requirements that generic components impose on their types:
  - Syntactic properties: e.g. types, name declarations
  - Semantic properties: e.g. axioms
  - Complexity guarantees
- Concepts are an essential component of generic programming.
  - Improve safe and efficiency-preserving reusability of software components
  - Abstract over properties of types, rather than types
- Concepts are constraints on types, which
- should be type-checked by the compiler
  - Improve error detection and diagnosis
  - Implementation encapsulation
- A language support for concepts could be useful for C++
  - E.g. Library developers and users.

Outline

① Concepts:

1. An Introduction
   - For generic programming, in C++
   - The elementary components
2. **Implementing Concepts**
   - **Implementation considerations**
   - **Design alternatives**
   - **Towards ConceptClang**

② ConceptClang: The Compilation Model

- The components
- Essential data structures and procedures
- Preliminary observations
- Case study: Type-checking (constrained) templates
- Open questions

Implementing Components

- ✧ Concept Definition:
  - Parse concept declaration
  - Name lookup:
    - Find names in refinements and associated requirements
  - Check body against parameters

```
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;

    requires IntegralType<difference_type>
                && Convertible<reference, const value_type &>
                && Convertible<pointer, const value_type*>;
    requires Convertible<HasDereference<postincrement_result>::result_type,
                         const value_type&>;

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

Implementing Components

- ✧ Concept Model (Template):
  - Parse concept model declaration
  - Name lookup:
    - Find names in modeled concept
    - Find names in maps of refinements and associated requirements
  - Check model against modeled concept:
    - Every requirement must be satisfied.
    - Every requirement satisfaction must correspond to a requirement.
    - A model must exist for every refinement.

```
template<typename T>
concept_map InputIterator<vector<T>::iterator> {
  //typedef value_type vector<T>::iterator::value_type;
  . . .
  pointer operator->(const vector<T>::iterator&) { … }
};

// Refinements and associated requirements are
// satisfied by defining models for them.
```

Implementing Components

✧ Constrained Template Definition

- Parse constraints specification
  - → **Constraints environment**.
  - → **Restricted scope**.
- Name lookup:
  - Find names in constraints environment
- Check body against constraints

```
template<typename I, typename T, typename Op>
  requires(InputIterator<I>, BinaryFunction<Op>,
           Assignable<I::value_type, Op::result_type>)
T accumulate(I first, I last, T init, Op bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Implementing Components

- ✧ Constrained Template Use
  - Constraints satisfaction
    - Finds models for each constraint, based on template arguments
  - Code generation (instantiation):
    - Rebuilds entity references, based on found models

```
vector<int> v;
int i = accumulate(v.begin(), v.end(), 0, plus<int>());
```

Type-checking Constrained Templates

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires(InputIterator<I>,
         BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Constrained Template Use:

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check Constraints-Check

INSTANTIATION

Code Generation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op)
{ ... }
```

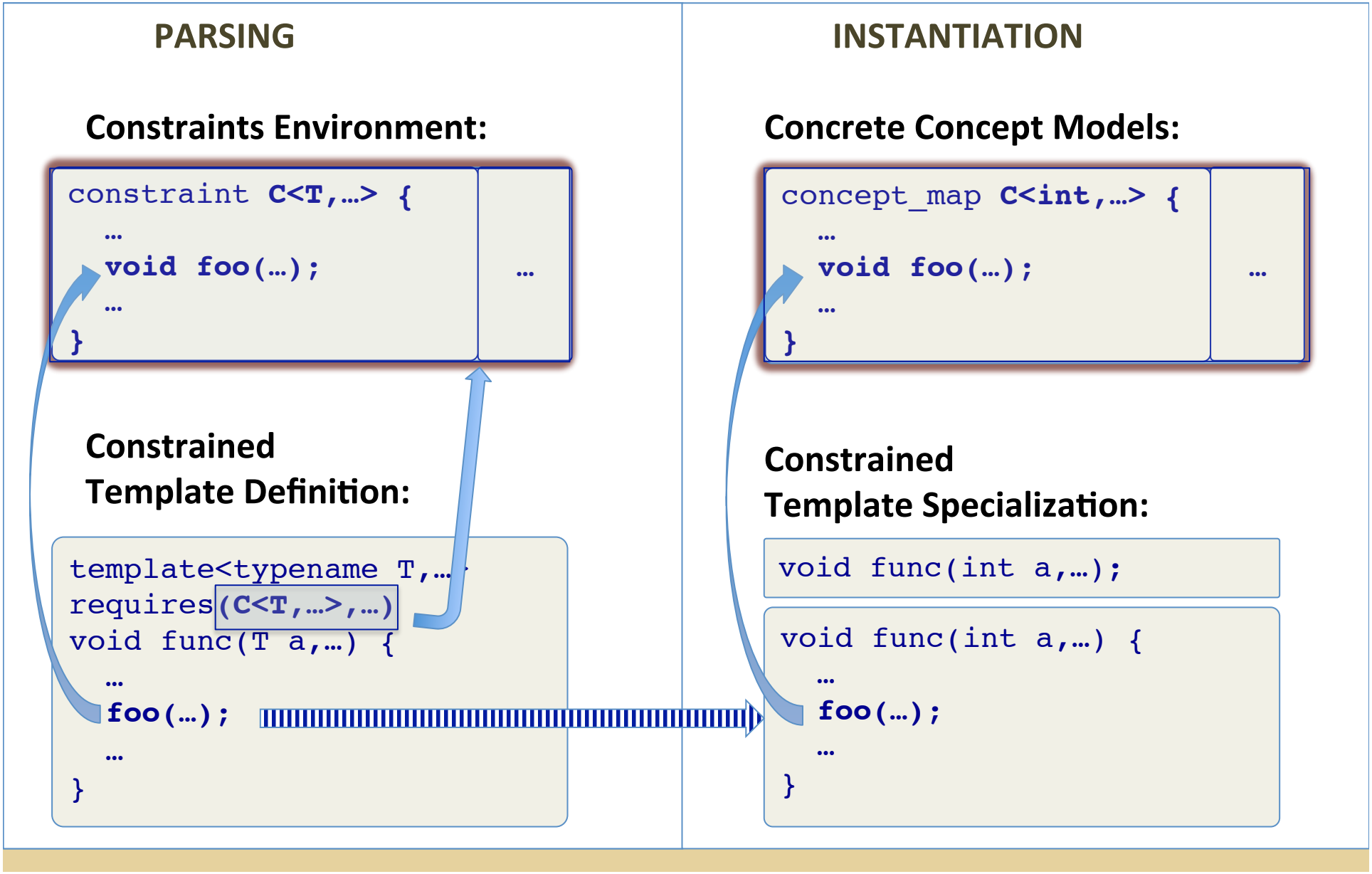
Once!

Specialization **+ Models**:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

Entity (Reference) Rebuilding: Example



Implementation Considerations

✧ Concept Definition:

- Parsing
- Name lookup, modified
  - In refinements
- Well-formed ?

✧ Constrained Template Definition:

- Parsing constraints environment
  - Restricted Scope:
    Starts at the beginning of the constraints environment
- Name lookup, modified
- Check body against constraints

✧ Concept Model (Template):

- Parsing
- Name lookup, modified
  - In refinement models
- Concept model checking
  - Refinement models ?
  - Requirement satisfactions ?

✧ Constrained Template Use:

- Constraints satisfaction
  - Storage for the models ?
- Transfer models to code generation.
  - Rebuilding entity references ?

Design Alternatives: Salient Differences

- Requirements representation

  When parsing concept definitions

- Modeling mechanism

  Match types to concepts

- Requirements satisfaction

  When checking concept models

- Checking body of constrained template definitions

  When parsing constrained template definitions

- Axioms representation and satisfaction
  - Usually parsed, but not checked
  - What logical sentences are expressible ?

Design Alternatives: Salient Differences

- Requirements representation
- Modeling mechanism
  - **Implicit** → User **cannot** explicitly define a concept model
    - Parsing of concept model declarations is not applicable.
    - Based on **structural conformance**.
  - **Explicit** → User **must** explicitly define a concept model
    - Exception: models of refinements
    - Based on **named conformance**.
- Requirements satisfaction
- Checking body of constrained template definitions
- Axioms representation and satisfaction

Design Alternatives: Salient Differences

- Requirements representation
  - Use-patterns
  - Pseudo-signatures
- Modeling mechanism
- Requirements satisfaction
  - Find a valid expression
  - Collect valid candidates
- Checking body of constrained template definitions
  - Match expression trees against use-patterns
  - Based on name lookup in constrained context
- Axioms representation and satisfaction

Use-Patterns

== expressions serving as pattern for valid uses of declarations.

EXAMPLE	
`*i++`	→ Valid uses of **operators * and ++** must match the pattern.
`T{exp}`	→ type of **exp** explicitly converts to type **T**.
`T={exp}`	→ type of **exp** implicitly converts to type **T**.
`T=={exp}`	→ type of **exp** exactly matches type **T**.

- Satisfaction (checking concept models)
  - Find a valid expression
- Checking body of constrained template definition
  - Match expression trees against use-patterns
    - The use-patterns are in concepts specified by the constraints.

Pseudo-signatures

== type signatures serving as mockups for multiple valid declarations.

EXAMPLE

	Possible matches for type `t`:
`T foo(T)`	`const t foo(t), t& foo(t),`
	`t t::foo(), …`
`typename T`	`typedef T t`

- Satisfaction (checking concept models)
  - Collect valid candidates
- Checking body of constrained template definition:
  - Based on name lookup in constrained context
    - All uses of declarations found in constraints are checked.

Design Alternatives: Salient Differences

- Requirements representation

  e.g. use-patterns, pseudo-signatures.

- Modeling mechanism

  e.g. implicit, explicit.

- Requirements satisfaction

  e.g. a valid expression, valid candidates.

- Checking body of constrained template definitions

  e.g. matching expression trees,

  based on name lookup in constraints.

- Axioms representation and satisfaction

  e.g. (extended) logical expressions

Design Alternatives

Proposed Design	Requirements Representation	Modeling Mechanism	Requirements Satisfaction	Checking Body of Constrained Template Defns	Axioms
Explicit Concepts	Pseudo-signatures	Explicit	Collect valid candidates	Based on name lookup in constraints environment	No
Implicit Concepts	Use-patterns	Implicit	Find a valid expression	Match expression trees against use-patterns	Yes, ext'd.
Pre-Frankfurt Design	Pseudo-signatures	**Both** Explicit and Implicit	Collect valid candidates	Based on name lookup in constraints environment	Yes.
Palo Alto Design	Use-patterns, ext'd with type annotations	Implicit	Find a valid expression	Match expression trees against use-patterns	Yes, ext'd.

Other Design Idioms

- Named concept models
  - Magne Haveraaen. 2007. Institutions, property-aware programming and testing. In Proceedings of the 2007 Symposium on Library-Centric Software Design (LCSD '07). ACM, New York, NY, USA, 21-30. DOI=10.1145/1512762.1512765 http://doi.acm.org/10.1145/1512762.1512765

- Axioms for compiler optimizations
  - Xiaolong Tang and Jaakko Järvi. Concept-based optimization. In ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07), October 2007

Infrastructure Design

Proposed Design	Requirements Representation	Modeling Mechanism	Requirements Satisfaction	Checking Body of Constrained Template Defns
Explicit Concepts	Pseudo-signatures	Explicit	Collect valid candidates	Based on name lookup in constraints environment
Implicit Concepts	Use-patterns	Implicit	Find a valid expression	Match expression trees against use-patterns
Pre-Frankfurt Design	Pseudo-signatures	**Both** Explicit and Implicit	Collect valid candidates	Based on name lookup in constraints environment
Palo Alto Design	Use-patterns, ext'd 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 environment**

Infrastructure Extensions

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 environment
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 environment**
Pre-Frankfurt Design	• Reuse Clang's • **1 new kind**	–	–	–
Palo Alto Design	• **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**

Outline

① Concepts:

1. An Introduction
   - For generic programming, in C++
   - The elementary components
2. Implementing Concepts
   - Implementation considerations
   - Design alternatives
   - Towards ConceptClang

② ConceptClang: The Compilation Model

- **The components**
- **Essential data structures and procedures**
- **Preliminary observations**
- **Case study: Type-checking (constrained) templates**
- **Open questions**

ConceptClang Objective

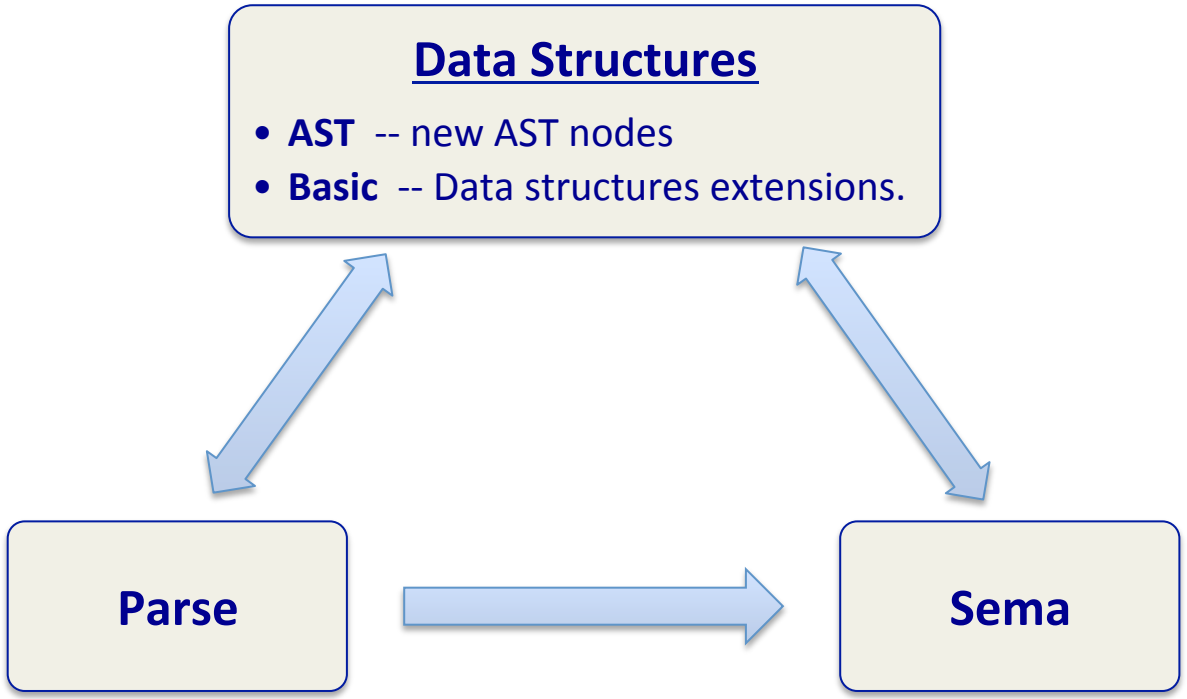
1. Define an infrastructure layer
   - Independent from any design alternative
   - Implementation separated from Clang
2. Extend infrastructure with design-specific implementations
   - TR **N2914=09-0104** -- a.k.a. "**pre-Frankfurt**" design
   - TR **N3351=12-0041** -- a.k.a "**Palo Alto**" design
3. Explore implications on the C++ language
   - Extensions from Clang
4. Explore (subtle) differences between design alternatives
   - Extensions from ConceptClang infrastructure

Talk Section Focus

1. Define an infrastructure layer
   - Independent from any design alternative
   - Implementation separated from Clang
2. Extend infrastructure with design-specific implementations
   - TR N2914=09-0104 -- a.k.a. “pre-Frankfurt” design
   - TR N3351=12-0041 -- a.k.a “Palo Alto” design
3. **Explore implications on the C++ language**
   - **Extensions from Clang**
4. **Explore (subtle) differences between design alternatives**
   - **Extensions from ConceptClang infrastructure**
5. **Case Study: Type-checking (constrained) templates**

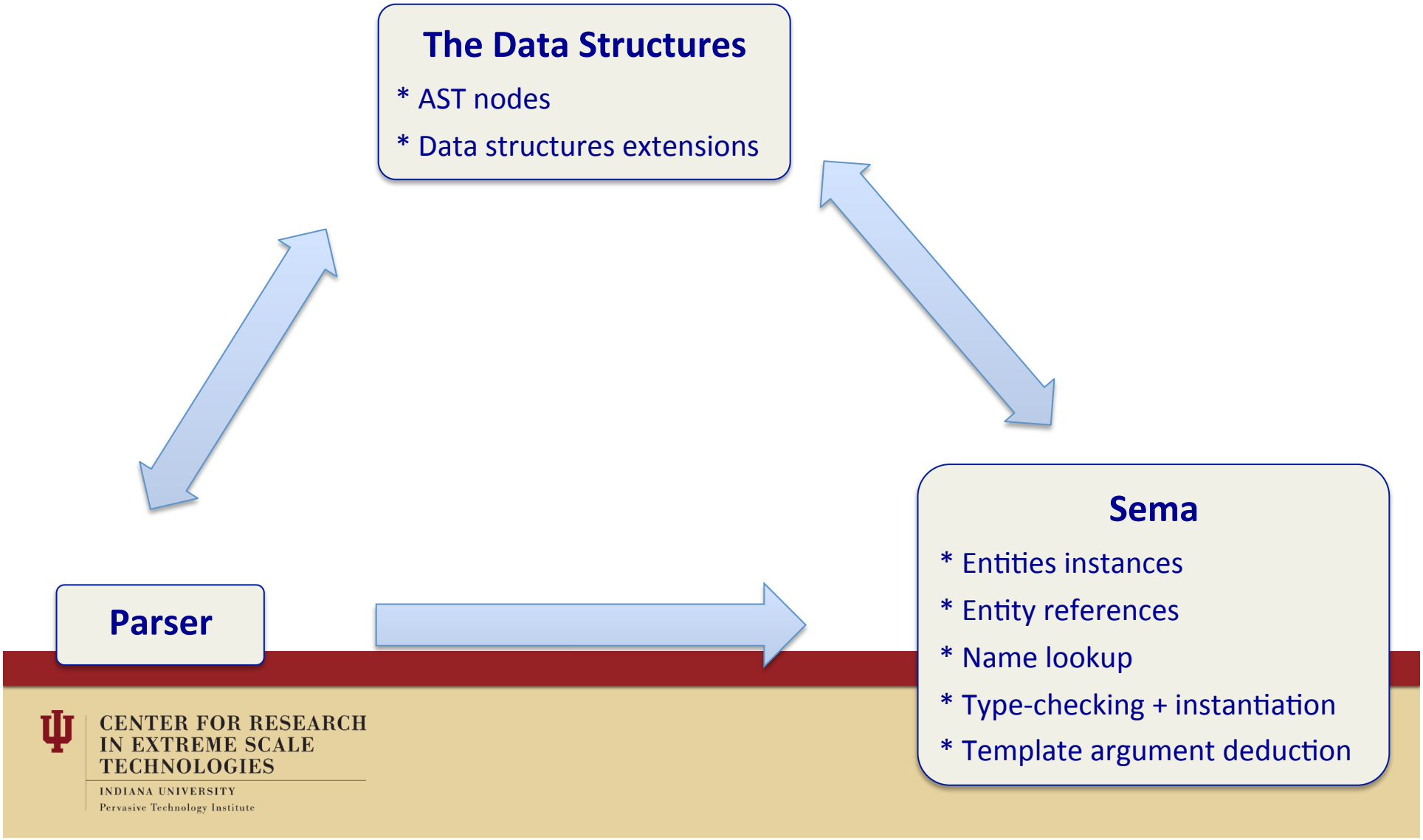
Clang → ConceptClang: The Components

- ConceptClang extensions affect only 4/~17 components of Clang:



- The **Driver** component is modified only for compiler flags support.
  - So, we do not consider it in our analysis.

The ConceptClang Components



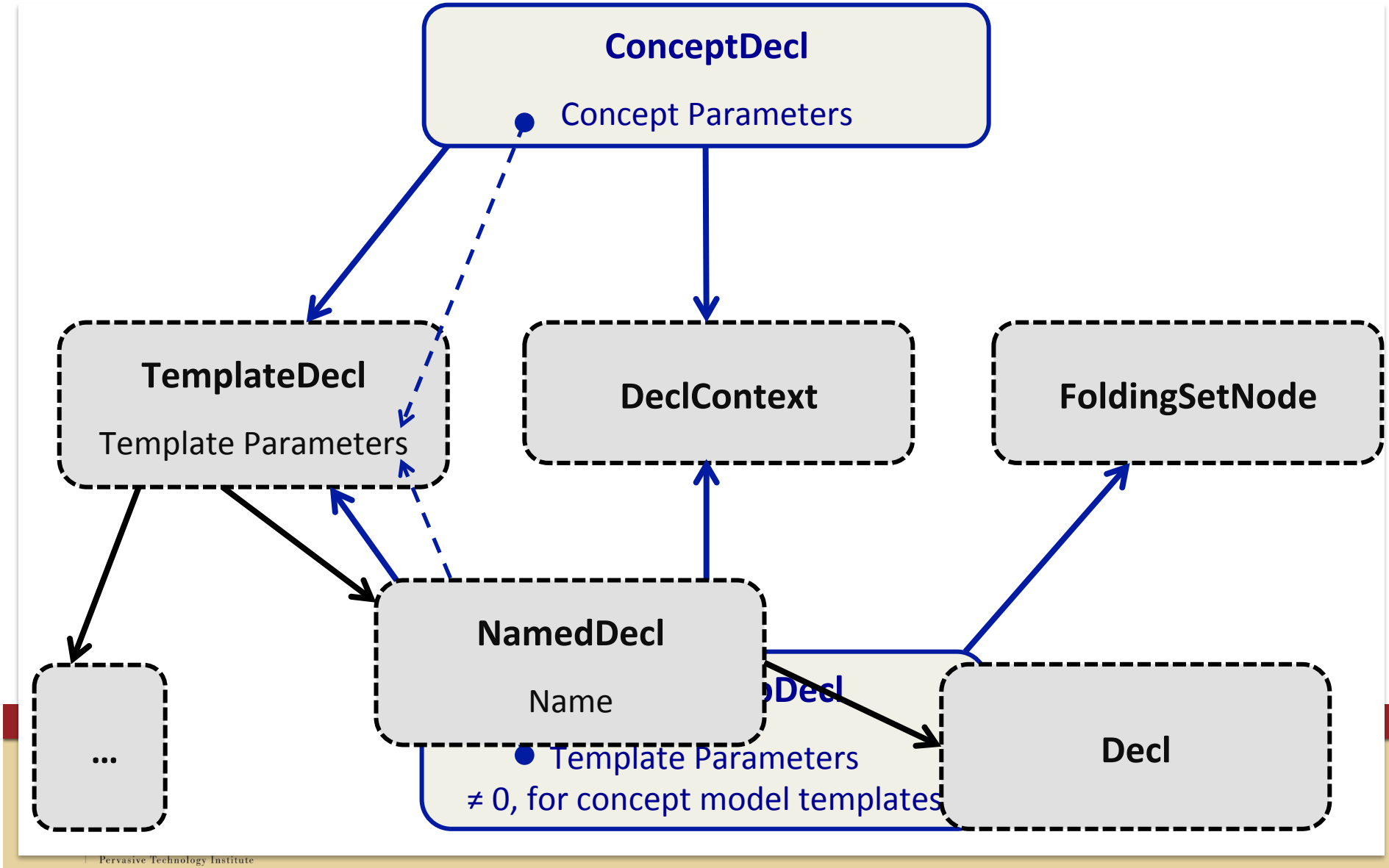
Infrastructure and Extensions

Proposed Design	**Requirements Representation**	**Modeling Mechanism**	**Requirements Satisfaction**	**Checking Body of Constrained Template Defns**
ConceptClang Infrastructure	**Declarations** [extend class **Decl**]	**Both Explicit and Implicit**	**Collect valid candidates**	**Based on name lookup in constraints environment**
Pre-Frankfurt Design	• Reuse Clang's • **1 new kind**	–	–	–
Palo Alto Design	• **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**

Essential Data Structures

- Concept definition declaration: class **ConceptDecl**.
  - Is a template declaration and declaration context
  - Requirements are represented as declarations
  - Instance holds a set of all its models.
- Concept model declaration: class **ConceptMapDecl**.
  - Is a template declaration and declaration context
  - Is uniquely identified for storage in sets

Clang -> ConceptClang: Data Structures



Essential Data Structures

- Concept definition declaration: class **ConceptDecl**
  - Is a template declaration and declaration context.
  - Requirements are represented as declarations.
  - Instance holds a set of all its models.
- Concept model declaration: class **ConceptMapDecl**
  - Is a template declaration and declaration context.
  - Is uniquely identified for storage in sets.
  - Requirements satisfaction **collect valid candidates**.
    - Candidates == declarations matching names or references in the requirement.
    - Add candidate declarations to the model declaration.

EXAMPLE

Requirement	Satisfied with
`*i++`	valid candidates for call **to operators * and ++.**
`T foo(T)`	valid candidates for call **`foo(a)`.**

Essential Data Structures

- **Concept model archetype**:
  == Fake concept model with minimal information.
  - Used as "placeholders" for concrete concept models.
  - Reuses **ConceptMapDecl** class.
  - Contains type-substituted copies of requirement declarations
    - Instead of requirement satisfaction candidate declarations.
- Constraints specifications and refinements:

EXAMPLE

Concept C<typename T>	Archetype for C: C<my_type>
`T&={*i++}`	`my_type&={*i++}`
`T foo(T)`	`my_type foo(my_type)`

Pseudo-signature Parsing

- Parse associated types as template parameters
  - Reuses Clang's procedures and classes.
- Parse associated functions as functions
  - Reuses Clang's classes procedures and classes.
- Parse associated requirements as refinements
  - Stores in new declaration: class **ConstraintsDecl.**

```
typename T;
…
requires C<T>;
…
T foo(T);
…
```

Pseudo-signature Checking

Satisfaction (concrete concept model):

- Three-stage name lookup:
  1. In concept model
  2. In surrounding scope
  3. In modeled concept (default implementation)
- Collect valid candidates at each stage
  - Only one candidate per associated type.

```
typename T;
…
requires C<T>;
…
T foo(T);
…
```

Substitution (concept model archetype):

- Reuses Clang.

Use-patterns Parsing

- Parse as expression
  - Extends Clang's expression parsing
    - Type annotations extension, new expression nodes, new declaration contexts.
  - Stores in a declaration: class **UsePatternDecl**
- Assume used declarations are not defined
  - Assume used types are defined (for now).
- Name lookup generates a named dummy declaration for each used declaration

Grammar:

```
exp;
type{exp};
type={exp};
type=={exp};
```

In Axioms:

```
exp => exp;
exp <=> exp;
```

EXAMPLE

`my_type{*i++}` → **Generates dummy declarations** for operators `*` and `++`. **[Parsing]**

Use-patterns Checking

Satisfaction (concrete concept model):

- Type-substitute in concept arguments
  - For each used dummy declaration:
    - Collect valid candidates
      → Perform name lookup
    - Add to checked concept model
- Rebuild with substituted result

Grammar:

```
exp;
type{exp};
type={exp};
type=={exp};
```

In Axioms:

```
exp => exp;
exp <=> exp;
```

Substitution (concept model archetype):

- For each used dummy declaration:
  - Type-substitute dummy declaration
  - Add copy to checked concept model

EXAMPLE

`my_type{*i++}`

→ **Generates dummy declarations** for operators `*` and `++`. **[Parsing]**
→ **Collect valid candidates** for calls to operators `*` and `++`. **[Satisfying]**
→ **Type-substitute dummies** for operators `*` and `++`. **[Substituting]**

Use-patterns Checking

Satisfaction (concrete concept model):

- Type-substitute in concept arguments
  - For each used dummy declaration:
    - Collect valid candidates
    - Add to checked concept model
- Rebuild with substituted result

Substitution (concept model archetype):

- For each used dummy declaration:
  - Type-substitute dummy declaration
  - Add copy to checked concept model

→ Type-substitutions extend code generation (instantiation)

- Special treatment for dummy declarations

Grammar:

```
exp;
type{exp};
type={exp};
type=={exp};
```

In Axioms:

```
exp => exp;
exp <=> exp;
```

Constrained Template Definition Parsing

- Constraints environment == concept model archetypes.
- Two-stage name lookup:
  - Interleaved with two-stage checking of entity references

Two-stage Entity Reference Checking

→ Two groups of entity references:

1. Entity is in restricted scope:
   - **Associated to a concept**
   - In template parameter scope

EXAMPLE: FAKE_ADD

```
template<InputIterator I, typename T>
T fake_add(I first, I last, T init) {
  for (; first != last; ++first)
    init = plus<T>()(init, *first);
  return init;
}
```

2. Entity is out of restricted scope:
   - **A constrained template**
   - reference is non-dependent

EXAMPLE: FAKE_ADD

```
template<InputIterator I, typename T>
T fake_add(I first, I last, T init) {
  for (; first != last; ++first)
    init = plus<T>()(init, *first);
  return init;
}
```

Constrained Template Definition Parsing

- Constraints environment == concept model archetypes.
- Two-stage name lookup:

1. In restricted scope
   == Lookup in model archetypes.
   - Check related entity reference

EXAMPLE: FAKE_ADD

```
template<InputIterator I, typename T>
T fake_add(I first, I last, T init) {
  for (; first != last; ++first)
    init = plus<T>()(init, *first);
  return init;
}
```

2. In outer scope of restricted scope
   - Check related entity reference

EXAMPLE: FAKE_ADD

```
template<InputIterator I, typename T>
T fake_add(I first, I last, T init) {
  for (; first != last; ++first)
    init = plus<T>()(init, *first);
  return init;
}
```

Constrained Template Definition Parsing

- Two-stage name lookup:

  1. In restricted scope

     == Lookup in model archetypes.

     - Check related entity reference

  2. In outer scope of restricted scope

EXAMPLE: FAKE_ADD

```
template<InputIterator I, typename T>
T fake_add(I first, I last, T init) {
  for (; first != last; ++first)
    init = plus<T>()(init, *first);
  return init;
}
```

- Palo Alto design extends Stage 1:
  - Expressions are marked for validation, as necessary
    - After successful checking of entity reference.
  - Marked expressions are checked against use-patterns in the constraints

    == **Expression Validation**

Essential Procedures

✧ Concept Definition:

- Name lookup in restricted scope
- Refinement parsing
- Requirement parsing

✧ Constrained Template Definition:

- Constraints specification parsing
- Name lookup in restricted scope
- Expression validation

✧ Concept Model:

- Name lookup in restricted scope
- Refinement satisfaction
- Requirement satisfaction:
  - Collect valid candidates
- Requirement substitution
- Concept model checking
- Model generation from model template
  - Requirement building

✧ Constrained Template Use:

- Constraints satisfaction
  - Concept model lookup
- Entity or reference rebuilding

Essential Procedures: Infrastructure

✧ Concept Definition:

- **Name lookup in restricted scope**
- **Refinement parsing**
- Requirement parsing

✧ Concept Model:

- **Name lookup in restricted scope**
- **Refinement satisfaction**
- Requirement satisfaction:
  - Collect valid candidates
- Requirement substitution
- **Concept model checking**
- **Model generation from model template**
  - Requirement building

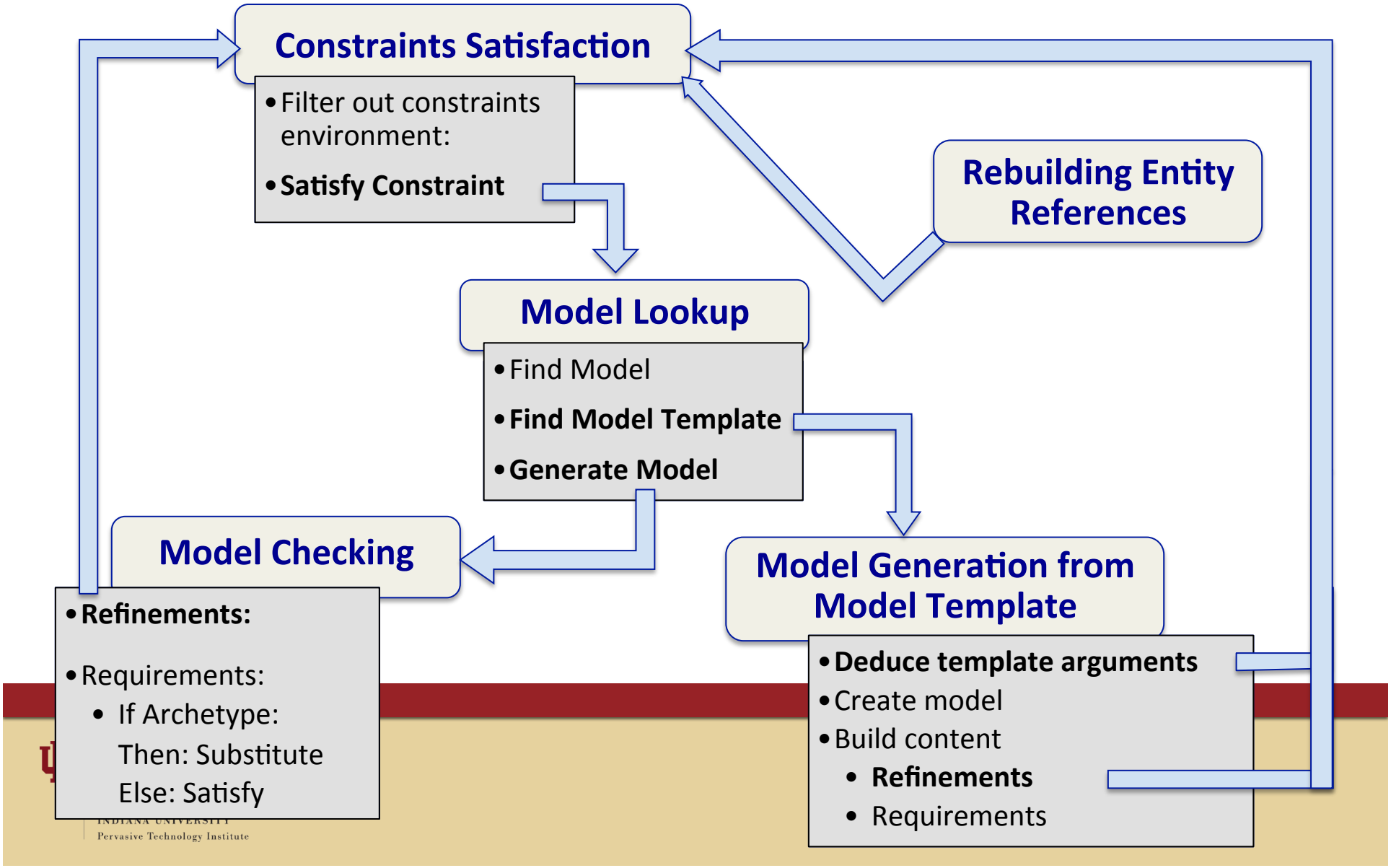
✧ Constrained Template Definition:

- **Constraints specification parsing**
- **Name lookup in restricted scope**
- ~~Expression validation~~

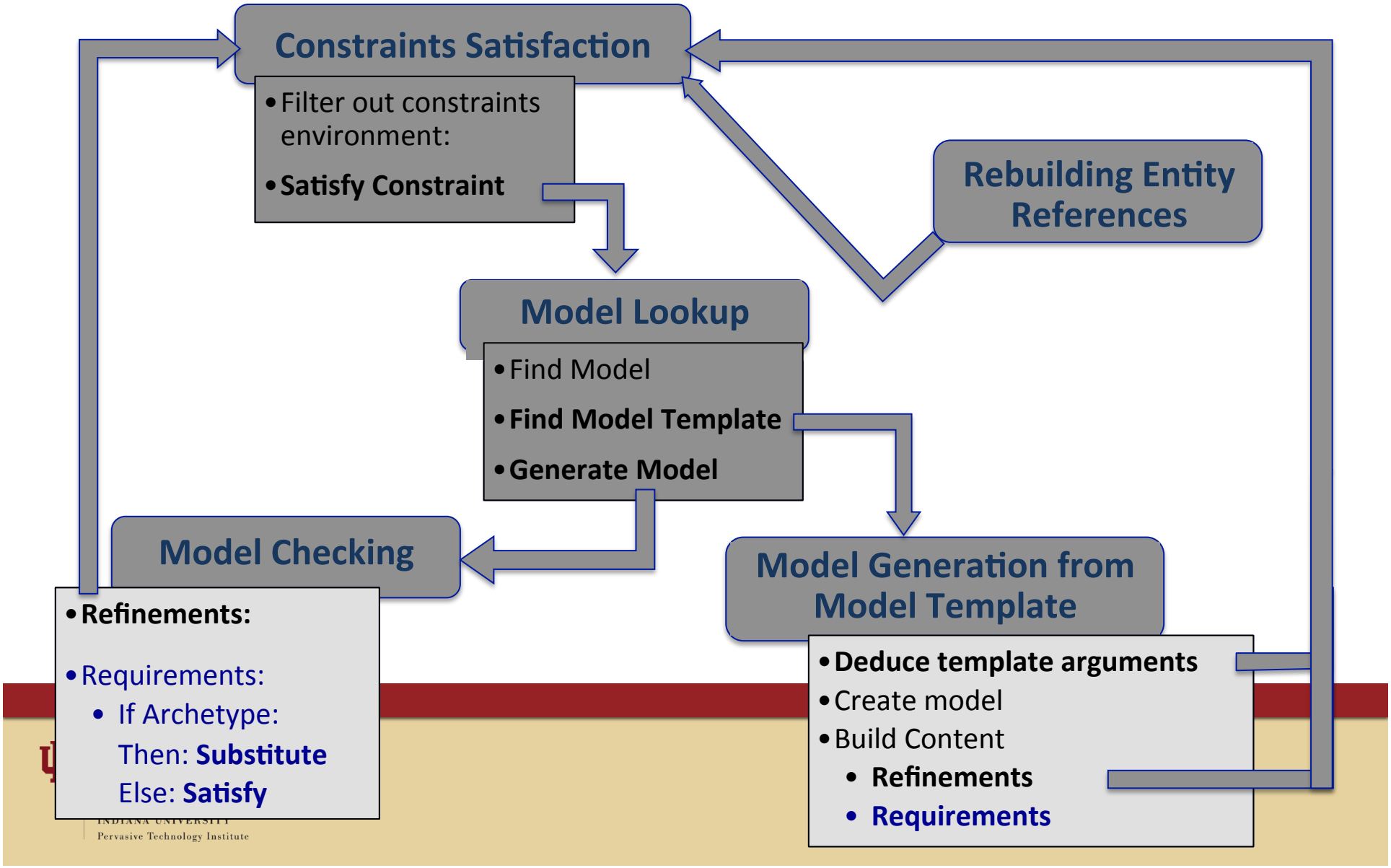
✧ Constrained Template Use:

- **Constraints satisfaction**
  - **Concept model lookup**
- **Entity or reference rebuilding**

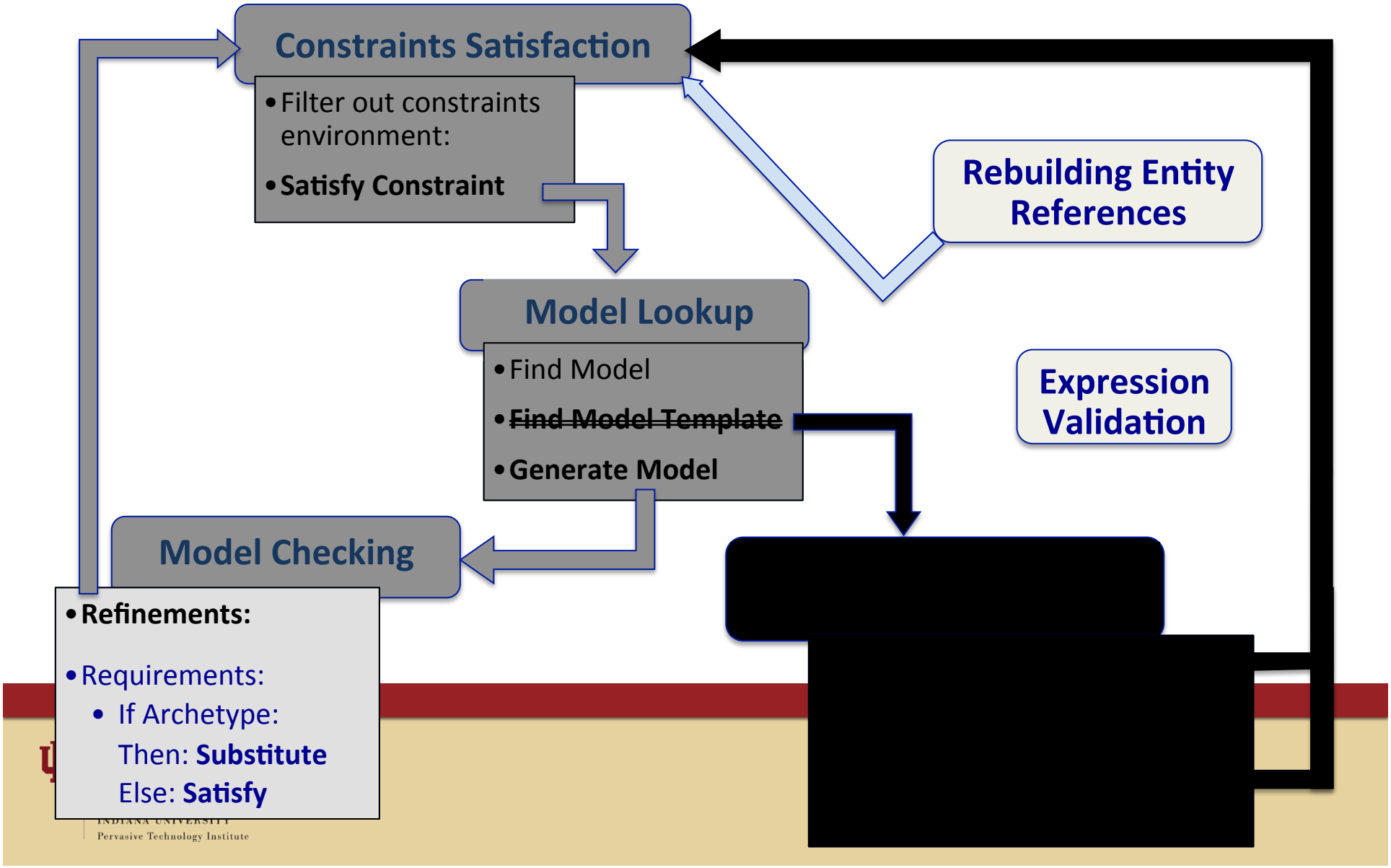
ConceptClang Infrastructure: Sema



Pre-Frankfurt Instantiation: Sema



Palo Alto Instantiation: Sema



Essential Procedures: Palo Alto

✧ Concept Definition:

- **Name lookup in restricted scope**
  - **generate dummy declarations**
- Refinement parsing
- **Requirement parsing**
  - **Types are visible in the surrounding context**
  - **Extended expressions parsing**

✧ Constrained Template Definition:

- Constraints specification parsing
- Name lookup in restricted scope
- **Mark expressions for validation**
- **Expression validation**
  - **Check expression tree against use-patterns in constraints**

✧ Concept Model:

- Name lookup in restricted scope
- Refinement satisfaction
- **Requirement satisfaction:**
  - **Type-substitute + Rebuild**
  - **Collect valid candidates**
- **Requirement substitution**
  - **Copy dummy declarations**
- Concept model checking
- ~~**Model generation from template**~~

✧ Constrained Template Use:

- Constraints satisfaction
  - Concept model lookup
- Entity or reference rebuilding
  - **Extended for requirement satisfaction**

Essential Procedures: Pre-Frankfurt

✧ Concept Definition:

- Name lookup in restricted scope
- Refinement parsing
- **Requirement parsing**
- **Associated types as template parameters**

✧ Concept Model:

- Name lookup in restricted scope
- Refinement satisfaction
- **Requirement satisfaction:**
  - **Three-stage name lookup**
  - **Collect valid candidates**
- **Requirement substitution**
- Concept model checking
- Model generation from model template
  - **Requirement building**

✧ Constrained Template Definition:

- Constraints specification parsing
- Name lookup in restricted scope

✧ Constrained Template Use:

- Constraints satisfaction
  - Concept model lookup
- Entity or reference rebuilding

A Summary of Differences: Parser

	Pre-Frankfurt	Palo Alto
New scopes entered after:	**‘{‘**	**‘=‘ and ‘{‘**
Scope of body of concept definition is also a template parameter scope:	**Yes**	No
Associated types as template parameters:	**Yes**	No
Extended expression parsing:	No	**Yes**
Parsed expressions are marked for validation:	No	**Yes**
Assumes that types are defined in the context of the definition of a concept:	No	**Yes**

A Summary of Differences: Structures

	Pre-Frankfurt	Palo Alto
New AST classes:	ConstraintsDecl	UsePatternDecl, UsePatternDummyExpr, DummyAssocDecl, RequiresDecl
Explicit concept models:	**Yes**	No
Concept model templates:	**Yes**	No

A Summary of Differences: Sema

	Pre-Frankfurt	Palo Alto
Name lookup generates DummyAssocDecls:	No	**Yes**
Concept overloading is allowed:	No	**Yes**
Requirements satisfaction:	**Lookup*3**	**Type-substitute + rebuild**
Extends template instantiation mechanism:	No	**Yes**
Satisfies associated functions (special case):	**Yes**	No
Expression validation:	No	**Yes**
Rebuilds type references (at instantiation):	**Yes**	No
Type-checks concept model templates:	**Yes**	No
Generates concept models from templates:	**Yes**	No

Pre-Frankfurt Subtle Extensions

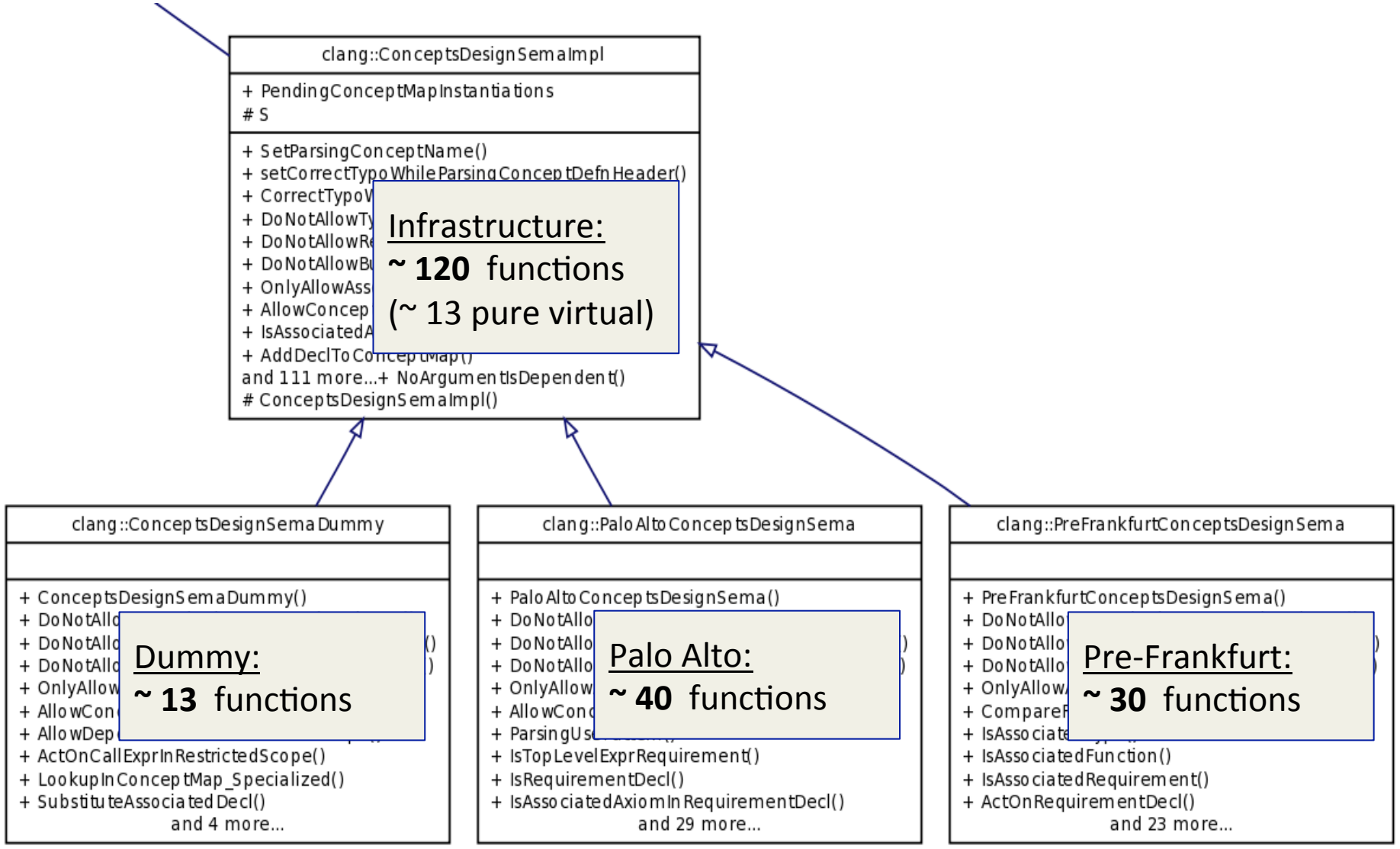
- Allow the shadowing of associated type declarations:
  - Only over associated type declarations of refinements
  - To override default implementations.

```
concept A<typename T> {
  typename my_type = …;
}
concept B<typename T> : A<T> {
  typename my_type = …;
}
```

- Allow the redefinition of pre-defined functions, including builtins:
  - Once, in the scope of concept definitions or models.

```
template<typename T>
T foo(T) { ... }

concept A<typename T> {
  T foo(T) { ... }
}
```

Implementation Workload: Sema



Clang → ConceptClang: A Sample Case

From:

Type-checking Templates, in Clang

To:

Type-checking Constrained Templates, in ConceptClang

Illustration with a function call expression

Type-checking Templates

PARSING

Template Definition:

```
template<typename Iter,
         typename T,
         typename BinOp>
T accumulate(...) { ... }
```

Template Use:

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check

INSTANTIATION

Code Generation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op)
{ ... }
```

Once!

Specialization:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

Type-checking Constrained Templates

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires(InputIterator<I>,
         BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Constrained Template Use:

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check Constraints-Check

INSTANTIATION

Code Generation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op)
{ ... }
```

Once!

Specialization **+ Models**:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

(Non)Dependent Entity References

- An entity reference: refers to an entity
  - e.g. A **function call expression**: refers to a function declaration
  - e.g. The **use of a type** to declare a variable
- A dependent entity reference:
  - depends on a template parameter
- A non-dependent entity reference:
  - does not depend on a template parameter

EXAMPLE: ACCUMULATE – ALL DEPENDENT ENTITY REFERENCES

```
template<typename InputIterator, typename T, typename BinaryFunction>
T accumulate(InputIterator first, InputIterator last,
             T init, BinaryFunction bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

(Non)Dependent Entity References

- An entity reference: refers to an entity
  - e.g. A **function call expression**: refers to a function declaration
  - e.g. The **use of a type** to declare a variable
- A dependent entity reference:
  - depends on a template parameter
- A non-dependent entity reference:
  - does not depend on a template parameter

EXAMPLE: ACCUMULATE_INT – SOME DEPENDENT ENTITY REFERENCES

```
template<template<typename T, typename A = allocator<T> >
         class Container>
int accumulate_int(typename Container<int>::iterator first,
                   typename Container<int>::iterator last,
                   int init, BinOp<int>& bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Templates: No Separate Type-checking

PARSING

Template Definition:

```
template<... class Container>
int accumulate_int(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Type-check non-dependent

Template Use:

Check arguments

```
vector<int> v;
int i = accumulate_int<vector>
(v.begin(), v.end(), 0, add_int);
```

INSTANTIATION

Code Generation:

```
int accumulate_int(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Type-check dependent

Specialization:

```
int accumulate_int
(vector<int>::iterator first,
 vector<int>::iterator last,
int init, BinOp<int>& bin_op);
```

Type-checking Constrained Templates

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires(InputIterator<I>,
         BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Constrained Template Use:

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check Constraints-Check

INSTANTIATION

Code Generation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op)
{ ... }
```

Once!

Specialization **+ Models**:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

Separate Type-checking ?

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires (InputIterator<I>,
          BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Type-check all

Constrained Template Use:

```
vector<int>
int i =
accum
                plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check arguments

Constraints-check arguments

INSTANTIATION

Code Generation:

```
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Specialization + Models:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

Not Quite Separate Type-checking

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires (InputIterator<I>,
          BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Type-check all

Constrained Template Use:

```
vector<int>
int i =
accum
                plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check arguments

Constraints-check arguments

INSTANTIATION

Code Generation:

```
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Potential ambiguities

Specialization + Models:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

Type-checking Templates

PARSING

Template Definition:

```
template<typename I, typename T,
         typename BinOp>
T accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  ret
}
```

Type-check non-dependent

Template Use:

```
vector<int>
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check arguments

INSTANTIATION

Code Generation:

```
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Type-check dependent

Specialization:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

Type-checking Constrained Templates

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires(InputIterator<I>,
         BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Type-check all

Constrained Template Use:

```
vector<int>
int i =
accum
                plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check arguments

Constraints-check arguments

INSTANTIATION

Code Generation:

```
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Handle ambiguities

Specialization + Models:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

Type-checking Templates in Clang

1. Parsing template definitions

2. Parsing template uses

3. Generating specialized implementations
   - At the end of parsing a translation unit.

Parsing A Template Definition in Clang

↳ Parse the template declaration

↳ Parse the body

↳ Parse the statements in the body

↳ Parse the expressions in each statement

`ParseTemplate.cpp`:
→ ParseTemplateDeclarationOrSpecialization()
↳ ParseSingleDeclarationAfterTemplate()

`ParseStmt.cpp`:
→ ParseStatementOrDeclaration()

```
template<...>
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Parsing A Template Use in Clang

↳ Parse the expressions in each statement:

↦ Parse a call expression:

↳ **Check call expression**

ParseExpr.cpp:
↳ ParseExpression()
↳ ParsePostfixExpressionSuffix()

↦ Parse a type reference:

↳ Check type reference

ParseExpr.cpp:
↳ ParseDeclarator()

```
template<...>
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

```
vector<int> v;
int i = accumulate(v.begin(),
                   v.end(), 0, plus<int>());
```

Checking A Call Expression in Clang

↳ Check call expression:

SemaExpr.cpp:
→ ActOnCallExpr()

↦ If dependent: Delay

↦ If direct:

```
template<...>
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

↦ If object or overloaded:

SemaOverload.cpp:
↦ BuildCallToObjectOfClassType()
↦ BuildCallToMemberFunction()
↦ **BuildOverloadedCallExpr()**

```
vector<int> v;
int i = accumulate(v.begin(),
        v.end(), 0, plus<int>());
```

Overload Resolution in Clang

↳ Overload resolution:

1. Build candidate sets

   ↳ **Template Argument Deduction**

SemaOverload.cpp:
→ BuildOverloadedCallExpr()

↦ AddFunctionOverloadcandidates()
↦ AddMemberOverloadCandidates()
↦ AddConversioncandidates()
↦ AddSurrogateCandidates()

2. Select best viable candidate

```
vector<int> v;
int i = accumulate(v.begin(),
        v.end(), 0, plus<int>());
```

Template Argument Deduction in Clang

↳ Template Argument Deduction:

SemaTemplateDeduction.cpp:
→ DeduceTemplateArguments()

1. Deduce the template arguments, if necessary
2. Check the template arguments
3. Create the template specialization

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

Creating A Template Specialization in Clang

↳ Create the template specialization:

SemaTemplateInstantiateDecl.cpp:
→ SubstDecl()

- Type-substitute the type signature
- Create a new **FunctionDecl** instance

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

Overload Resolution in Clang

↳ Overload Resolution:

1. Build Candidate Sets
   - ↳ **Template Argument Deduction**
2. Select best viable candidate
   - ↳ **Template Argument Deduction**
3. Mark for instantiation
4. Resolve

SemaOverload.cpp:
→ BuildOverloadedCallExpr()

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
vector<int> v;
int i = accumulate(v.begin(),
        v.end(), 0, plus<int>());
```

Type-checking Templates in Clang

1. Parsing template definitions
   1. Parsing template declarations
   2. Parsing template uses
2. Parsing template uses
   1. Parsing expressions
   2. Overload resolution
   3. Template argument deduction
   4. Creating template specialization
   5. Selecting best viable candidate
   6. Marking selected candidate for instantiation

```
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```


```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

3. **Generating specialized implementations**
   - At the end of parsing a translation unit.

Template Instantiation in Clang

↳ Instantiate Specialization:

- Body of template
  == pattern for body of specialization

SemaTemplateInstantiateDecl.cpp:
↳ InstantiateFunctionDefinition()

↳ Type-substitute the statements in the body

SemaTemplateInstantiate.cpp:
→ SubstStmt()

↳ Transform the statements in the body

SemaTemplateInstantiate.cpp:
TreeTransform.cpp:
→ TransformStmt()

↳ Transform entities in each statement

```
template<typename I, ...>
T accumulate(I first, ...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Transforming Entities in Clang

↳ Transform entities in each statement:

- ↦ Expressions
- ↦ Statements
- ↦ Declarations
- ↦ Types
- ↦ …
- ↦ **Call expressions**

`SemaTemplateInstantiate.cpp`:
`TreeTransform.cpp`:
↦ TransformExpr()
↦ TransformStmt()
↦ TransformDecl()
↦ TransformType()
↦ …
↦ **TransformCallExpr()**

- General idea:
  1. Type-substitute (components)
     - Pick up specialized implementations
  2. Return (rebuild)

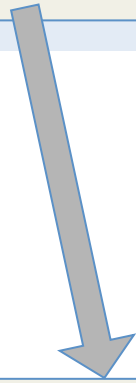
Transforming Entities in Clang

↳ Transform entities in each statement:

- ↦ Expressions
- ↦ Statements
- ↦ Declarations
- ↦ Types
- ↦ …
- ↦ **Call expressions**

SemaTemplateInstantiate.cpp:

```
template<typename I, ...>
T accumulate(I first, ...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```



```
int accumulate(…iterator first, ...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

- General idea:
  1. Type-substitute (components)
     - Pick up specialized implementations
  2. Return (rebuild)

Transforming Call Expressions in Clang

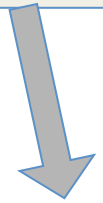
↳ Transform call expressions:

1. Transform sub-expressions
   - Callee
   - Call Arguments
2. Rebuild call expression

   == Check call expression
   - ↦ Resolve
   - ↦ Overload resolution

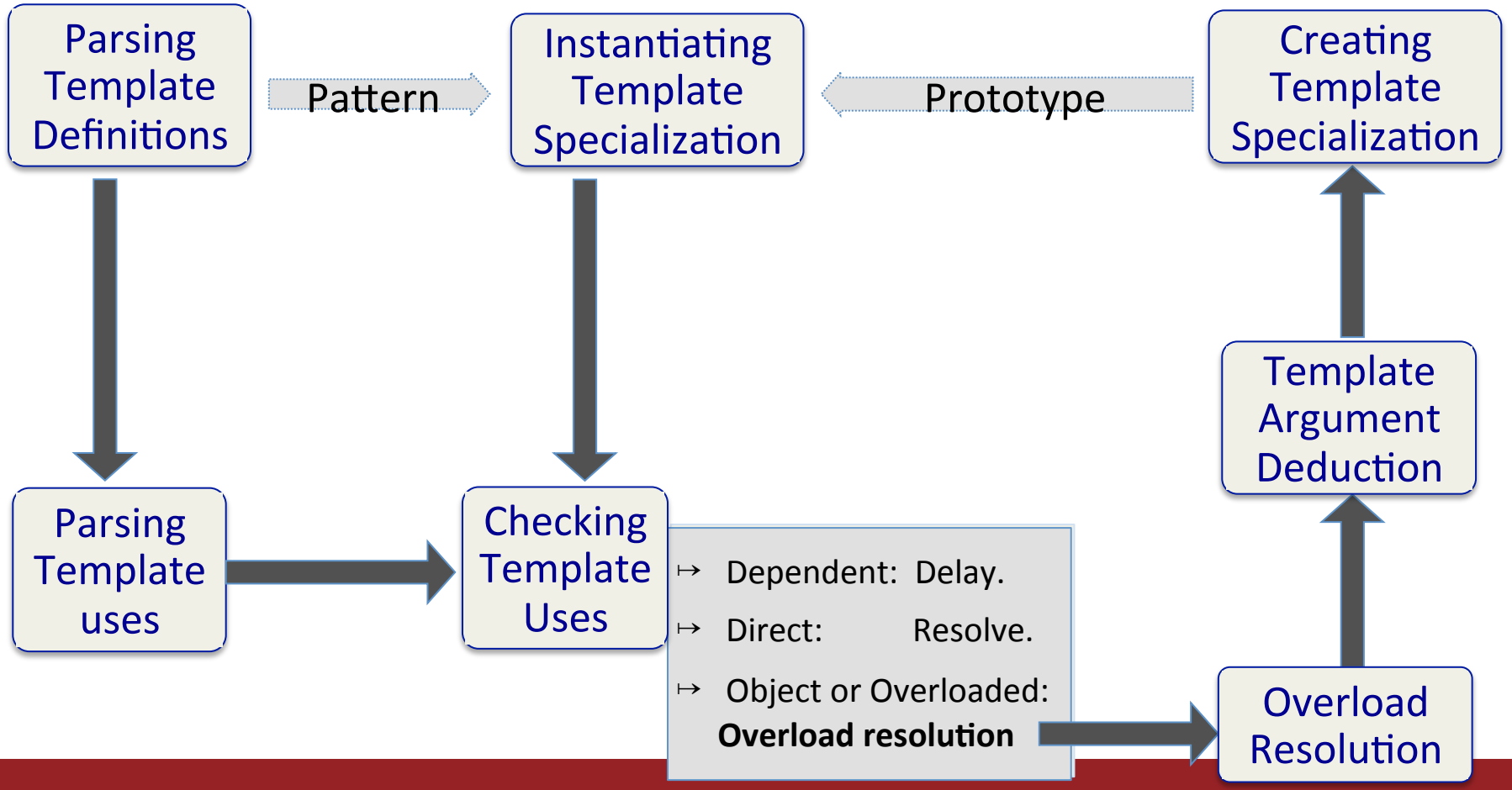
SemaTemplateInstantiate.cpp:
TreeTransform.cpp:
↦TransformCallExpr()

```
template<typename I, ...>
T accumulate(I first, ...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

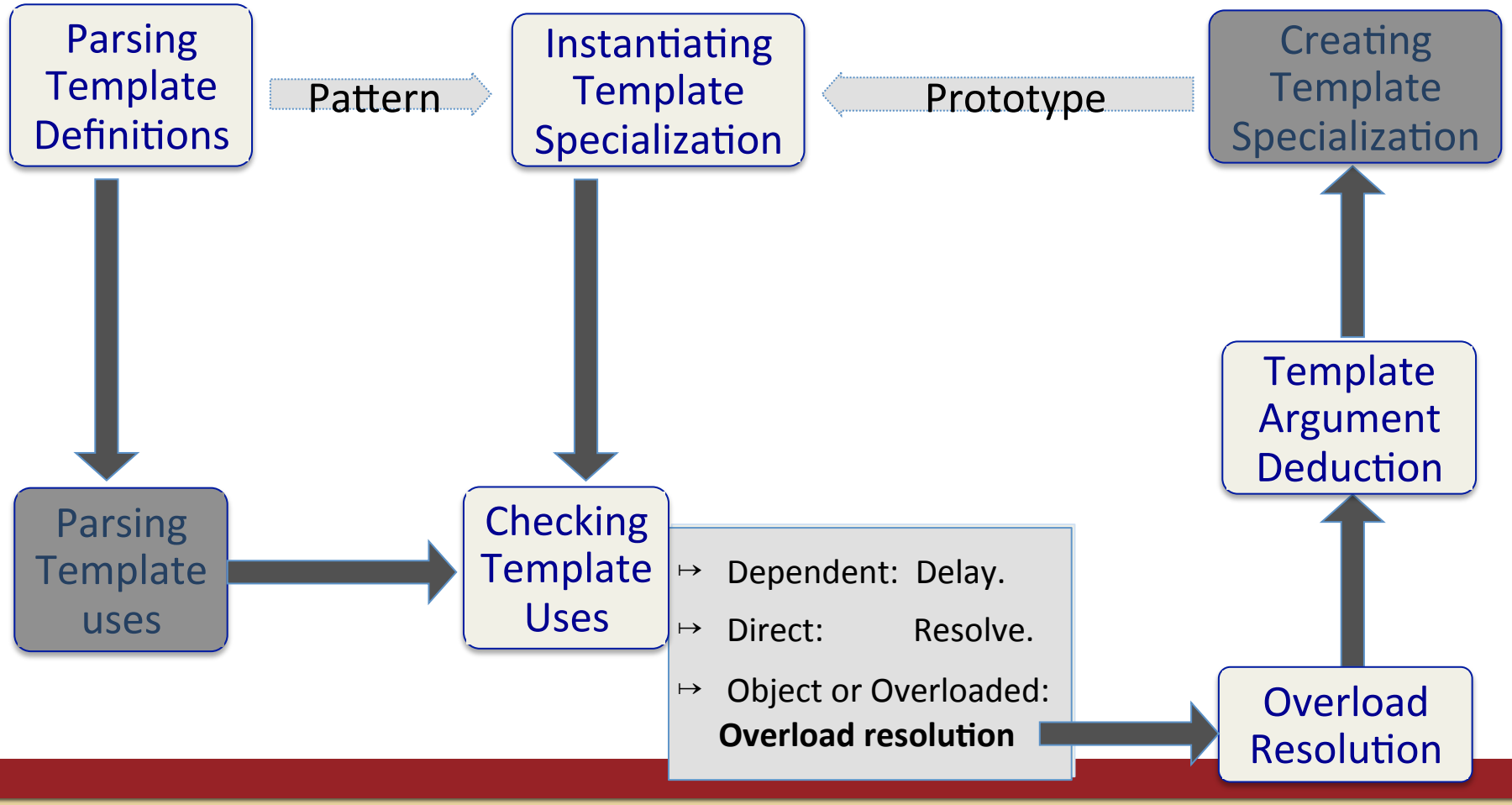


```
int accumulate(…iterator first, ...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Type-checking Templates in Clang



Type-checking Templates in ConceptClang



Type-checking Templates in ConceptClang

1. Parsing constrained template definitions

2. Checking template uses

3. Overload resolution and template argument deduction

4. Generating specialized implementations
   - At the end of parsing a translation unit.

Type-checking Templates in ConceptClang

1. Parsing constrained template definitions

   – Parsing constraints specifications
     - into concept model archetypes

   – Two-stage name lookup:
     1. Lookup in restricted scope
     2. Lookup in parent scope of outermost restricted scope

`ParseTemplate.cpp`:
→ ParseTemplateDeclarationOrSpecialization()
↳ ParseSingleDeclarationAfterTemplate()

`SemaLookup.cpp`:
→ CppLookupName()
↳ LookupQualifiedName()

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

Type-checking Templates in ConceptClang

2. Checking template uses
   - No delay of dependent entity references
   - Two-stage entity reference checking:
     1. Entities in restricted scope
     2. Entities out of restricted scope
        → Repeat from parent scope of outermost restricted scope

SemaExpr.cpp:
→ ActOnCallExpr()

Checking Template Uses [*2]

↦ ~~Dependent: Delay.~~

↦ Direct: Resolve.

↦ Object or Overloaded: **Overload resolution**

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

Type-checking Templates in ConceptClang

3. Overload resolution and template argument deduction

   1. Template argument deduction, extended

      - **Constraints satisfaction**

        generates models

`SematemplateDeduction.cpp`:
→ DeduceTemplateArguments()

1. Deduce the template arguments…
2. Check the template arguments
3. **Satisfy constraints**
4. Create the template specialization

   2. Selecting best viable candidate, extended

      - **Compare constraints**

Type-checking Templates in ConceptClang

4. Generating specialized implementations

Code transformations, extended

– **Entity or reference rebuilding, model-based**
  - w.r.t. models from constraints satisfaction.
  - Picks up implementation in the models.

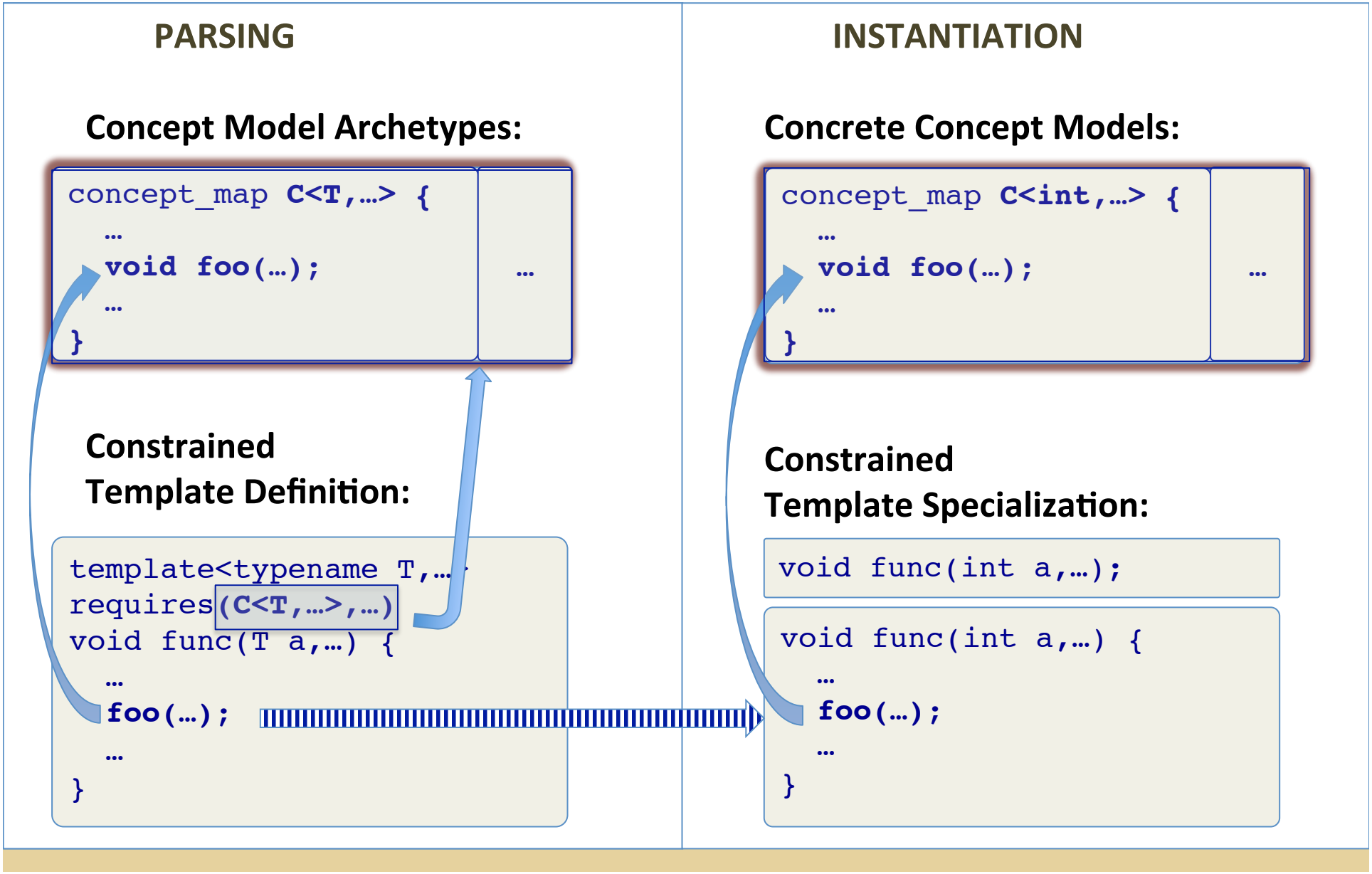
– Repeats the checking of template uses
  - i.e. ActOnCallExpr()

`SemaTemplateInstantiate.cpp`: `TreeTransform.cpp`

↦ Expressions: TransformExpr()
↦ Statements: TransformStmt()
↦ Declarations: TransformDecl()
↦ Types: TransformType()
↦ …
↦ **Call expressions: TransformCallExpr()**

- General idea:
  1. **Rebuild entity, model-based.**
  2. Type-substitute (components)
     - Pick up specialized implementations
  3. Return (rebuild)

Entity Reference Rebuilding: Example



Entity Reference Rebuilding: Example

- Entities associated to concepts:
  - Parsing constrained template definition:
    - **++** found in concept model archetype **InputIterator<I>**
  - Checking constrained template use:
    - Model **InputIterator<int>** is found
      - Model contains implementation of **++** for element type **int**.
  - Instantiating constrained template:
    - Reference to **++** must point to implementation in **InputIterator<int>**
    - Repeat name lookup
      - In model **InputIterator<int**
    - Rebuild entity reference
      - Potential ambiguity!
    - Mark model for instantiat

EXAMPLE: FAKE_ADD

```
template<InputIterator I, typename T>
T add(I first, I last, T init) {
  for (; first != last; ++first)
    init = plus<T>(init, *first);
  return init;
}
...
vector<int> v;
int i = add(v.begin(), v.end(), 0);
```

Updating Entity References: Example

- Constrained templates:
  - Parsing constrained template definition:
    - Constrained template **plus() :**

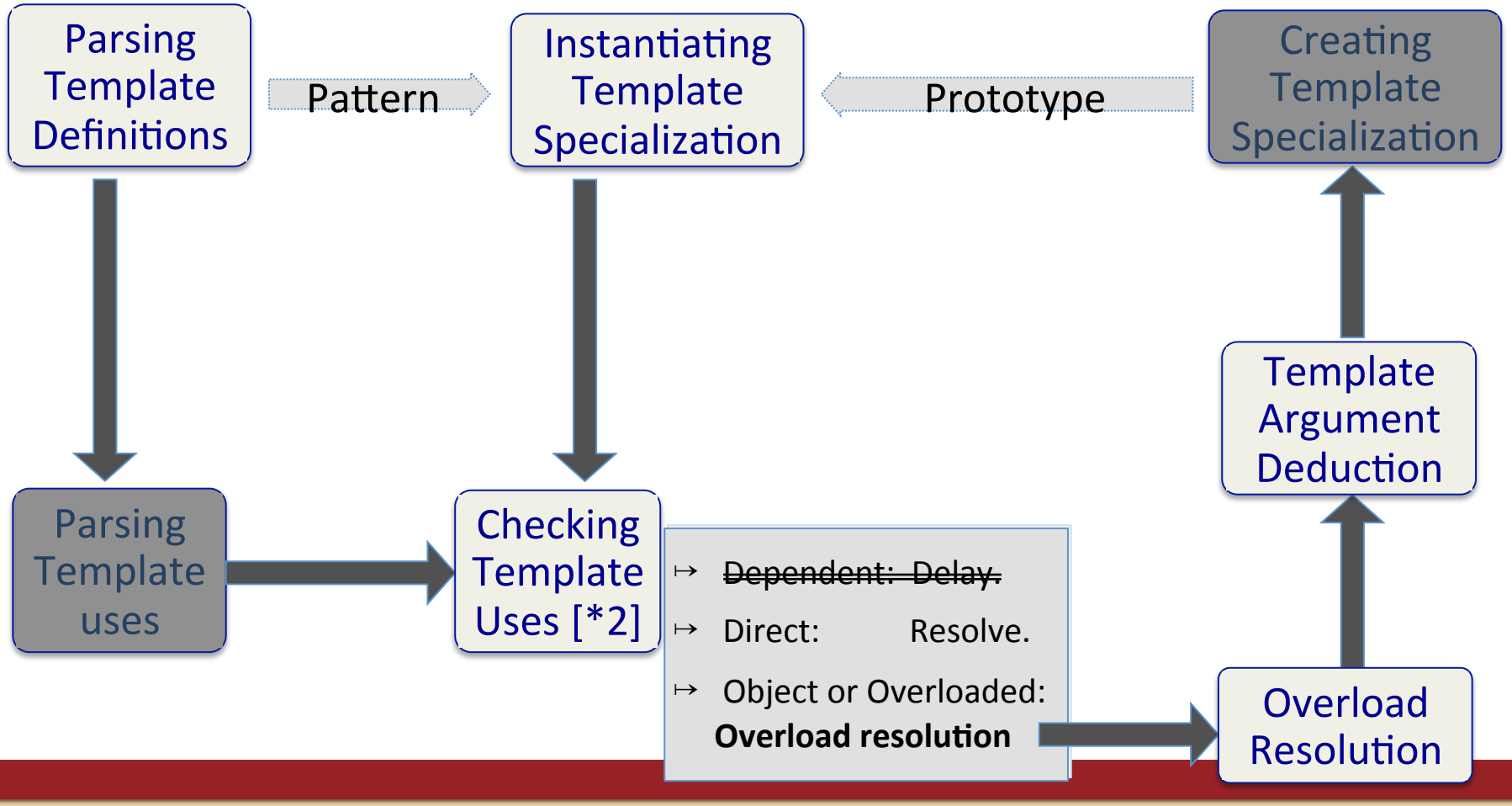
      Found in parent scope of outermost restricted scope.
    - Constraints satisfaction (in overload resolution):

      Generates "place-holder" model archetypes
  - Checking constrained template use:
    - N/A
  - Instantiating constrained tem
    - Reference to **plus()** must k
    - Repeat name lookup in su
      - Simply update "place-holde
    - Rebuild entity reference…
      - Potential ambiguity

EXAMPLE: FAKE_ADD

```
template<InputIterator I, typename T>
T add(I first, I last, T init) {
  for (; first != last; ++first)
    init = plus<T>(init, *first);
  return init;
}
...
vector<int> v;
int i = add(v.begin(), v.end(), 0);
```

Type-checking Templates in ConceptClang



Essential Procedures, Implemented

- ✧ Concept Definition
  - Name lookup in restricted scope
  - Refinement parsing
  - Requirement parsing
- ✧ Constrained Template Definition
  - Constraints specification parsing
    - Concept model archetypes
  - Name lookup in restricted scope
  - Expression validation
- ✧ Concept Model
  - Name lookup in restricted scope
  - Refinement satisfaction
  - Requirement satisfaction:
    - Collect valid candidates
  - Requirement substitution
  - Concept model checking
  - Model generation from model template
    - Requirement building
- ✧ Constrained Template Use
  - Constraints satisfaction
    - Concept model lookup
  - Entity or reference rebuilding
    - Extension for requirement satisfaction

Some General Design Observations

- Should concept parameter lists include constrained template parameters?
  - Not supported in Pre-Frankfurt design.
  - Supported in Palo Alto design and ConceptClang.

```
concept ForwardIterator
        <InputIterator I> { ... }

concept ForwardIterator
        <typename I>
    : InputIterator<I> { ... }
```

- Should concept model declarations be inserted into the scope in which they are created?
  - ConceptClang models are not visible by name lookup.

```
// Parse, insert C in scope.
concept_map C<my_type> { ... }

// Lookup(C) = {...,C<my_type> ?}
```

Some General Design Observations

- Is there any way to complete separate type-checking?
  - Open question.
- How to order template specializations w.r.t. constraints?
  - Open question: Concept-based overloading issues.
- …

Some pre-Frankfurt Design Questions

- Entering a new scope for concept definitions or models:

  at open brace or before concept refinements?

  - standard is unclear.
  - ConceptClang enters at open brace.

- Satisfying associated function requirements:

  How necessary is the generation of dummy expressions?

  - Facilitates the implicit deduction of values for associated types.
  - Currently not supported by ConceptClang.

Some Palo Alto Design Questions

- Could we gain from a lighter-weight approach to concepts?
  - e.g. Checking a constrained template body without binding dependent call expressions?
  - e.g. Assume all names are visible when parsing concept definitions?
- Could we gain from a heavier-weight approach to concepts?
  - e.g. Parse use patterns immediately into pseudo-signatures?
  - e.g. Make no assumption about name visibility when parsing concept definitions?

Recap: Objective

- Implement concepts for C++
  - in a modular and generic fashio

ConceptsDesign.{h,cpp}
Classes ConceptsDesign*Impl

- Define an abstract layer for reasoning about concepts designs.
  - Its implementation should be in
  - Its implementation should be ex

***ConceptsDesign.{h,cpp}**
Classes *ConceptsDesign*Impl

- Concretely investigate the implications of concepts designs on both implementation and usability
  - e.g. The implementation of con standard.

Underway… Our testing bedframe is still very minimal.
Connections to other C++ features need to be examined.

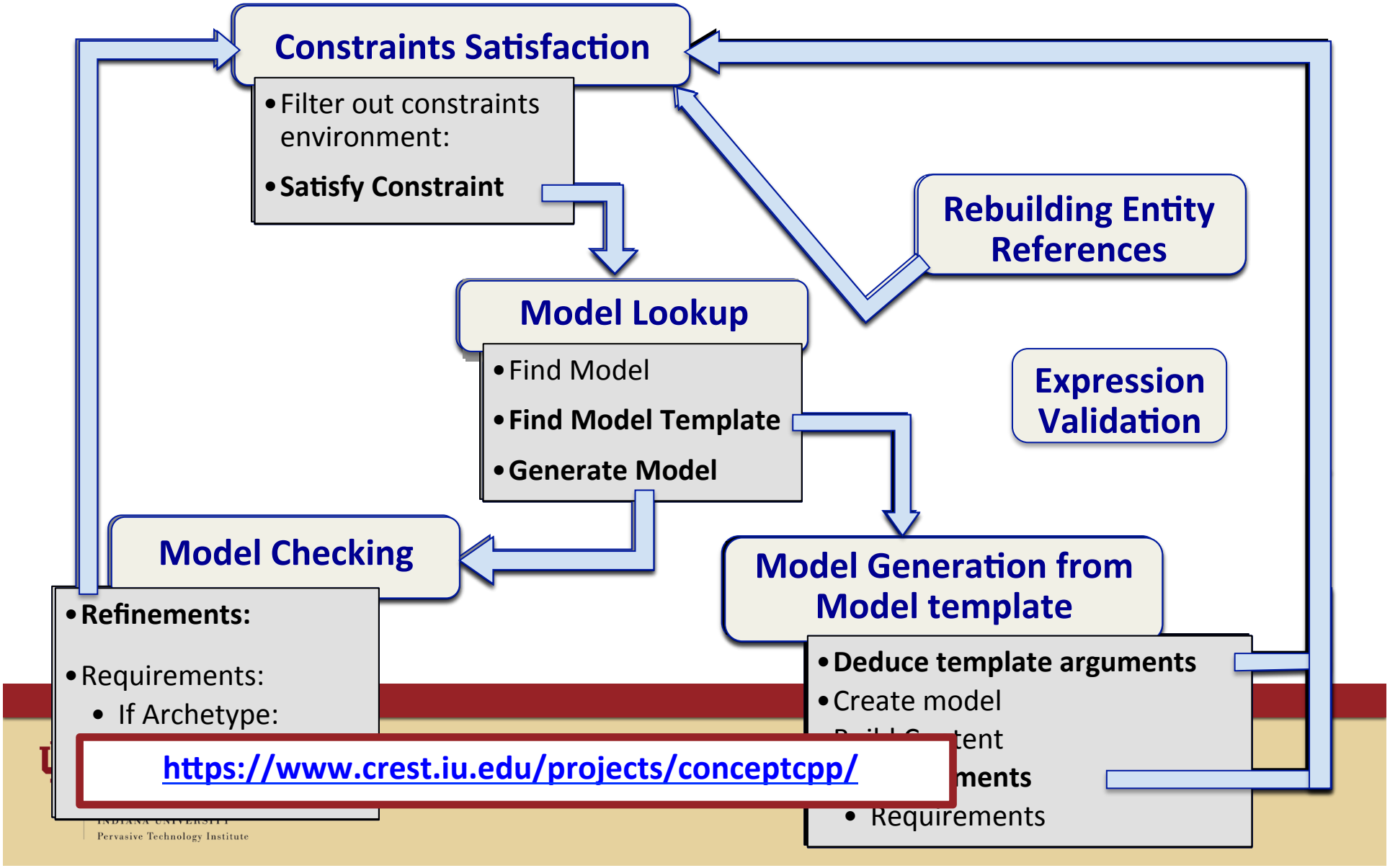
- Highlight the similarities and subtle differences between alternative designs
  - The implementation must clearl **instantiations** layer.

Pending concrete and throughout testing…

Conclusion

- Concepts
  - are an essential component of generic programming.
  - Improve the safety of C++ templates.
- ConceptClang
  - implements C++ concepts in Clang.
    - Clang facilitates the implementation.
  - is independent from alternative designs.
  - is extensible to alternative design.
  - highlights the (subtle) implications of design alternatives on the language mechanics.
  - is still under development and undergoing testing.

Thank You! Questions?



Concept-based overloading

- Function Overloading:
  - Partial order on functions
  - Select **most specialized** function
- Concept-based Overloading:
  - Constraints resolve ambiguities
  - Select **most constrained** as well
- **Question:**

  Given functions A and B, when:
  - **A** is **more constrained** than **B**, and
  - **B** is **more specialized** than **A**
  - **Select which one?**

```
//#1
```

```
//#1
template <class FI>
FI min_element(FI first, FI last);

//#2
template <class FI>
requires ForwardIterator<FI>
FI min_element(FI first, FI last);
```

```
//#B
template <class T>
T* min_element(T* first, T* last);

//#A
template <class FI>
requires ForwardIterator<FI>
FI min_element(FI first, FI last);

concept_map ForwardIterator<int*>;

//Instantiation chooses ???
int *p1, *p2;
int* p = min_element(p1, p2);
```

Concept-based overloading

- **Question:**

  Given functions A and B, when:
  - **A** is **more constrained** than **B**, and
  - **B** is **more specialized** than **A**
  - **Select which one?**

```
//#B
template <class T>
T* min_element(T* first, T* last);

//#A
template <class FI>
requires ForwardIterator<FI>
FI min_element(FI first, FI last);

concept_map ForwardIterator<int*>;

//Instantiation chooses ???
int *p1, *p2;
int* p = min_element(p1, p2);
```

- **Answer:**
  - Pre-Frankfurt C++: **B**
  - ConceptClang: **A (by default, options available)**
  - **Right answer?** **Open question!**

ConceptClang: Essential Procedures

Constraints Satisfaction:

- ✧ Takes In:
  - Template arguments -- @use
  - Template parameters -- @defn
  - Constraints on parameters -- @defn
  - Constraints environment -- @use
- ✧ Returns:
  - Concrete concept models
  - Concept model archetypes, only if
    - the constraints environment contains archetypes.
- ✧ Extends SFINAE:
  - Constraints satisfaction failure is a substitution failure.

The Procedure:

- For each constraint **C** on parameters:
  - Let **C** = type-substituted **C**.
  - If **C** has a copy **CC** in the constraints environment,
    - Add **CC** to the returned models.
  - Otherwise,
    - Satisfy **C**.
      - via **concept model lookup**.

ConceptClang: Essential Procedures

Concept Model Lookup:

- Try to find a model, or
- try to find a model template,
  - if applicable.
  - Generate a concrete model from it. Or
- generate a model,
  - If the mapped concept is considered implicit.
- Reverse the lookup result,
  - if the constraint is negative, and
  - if the result is not ambiguous

ConceptClang: Essential Procedures

Concept Model From Concept Model Template:

- Given:
  - the concrete concept model -- say **CM**'s arguments, and
  - the concept model template -- say **CMT**.
- Deduce the template arguments for **CMT**'s parameters.
  - The process satisfies the constraints on **CMT**'s parameters.
  - Let **IC_models** = the models generated.
- Create **CM**.
- Build the content of **CM** from that of **CMT**:
  - For refinements,
    - Reuse **constraints satisfaction**,
    - With **IC_models** as the constraints environment.
  - For requirements,
    - Use design-specific implementation

ConceptClang: Essential Procedures

Concept Model Checking:

- Check refinements:
  - Find or generate models
    - Reuses **constraints satisfaction**.
- Design-specific pre-processing function
  - Provided for genericity
- Check requirement declarations:
  - If the model is an archetype,
    - **Substitute** – design-specific
  - Otherwise,
    - **Satisfy** -- design-specific
- Design-specific post-processing function
  - Provided for genericity

Concept Model Template Checking:

- Follow concept model checking
- Use model archetypes as placeholders
  - whenever constraints satisfaction cannot find a concrete concept model.

ConceptClang: Essential Procedures

Rebuilding Entity References:

- Let **CS_models** = the result of constraints satisfaction.
- For an entity associated to a concept -- say **CD**:
  - Let **CM** = the concept model for **CD** in **CS_models**.
  - Find the entity's implementation in **CM**.
  - Rebuild the reference with the implementation.
- For a constrained template use,
  - rebuild the reference
  - with **CS_models** serving as the constraints environment.

ConceptClang: Essential Procedures

Expression Validation:

- Given:
  - An expression to validate – say **E**, and
  - The constraints environment – say **Cs_env**.
- Validate **E**'s sub-expressions with **Cs_env**.
  - Fail if the process fails for any sub-expression.
- Validate **E** against **Cs_env**:
  - For each use-pattern **U** in **Cs_env**:
    - Compare expression trees for **E** and **U**.
    - Return success when a match is found.
- If the validation fails:
  - Rebuild **E** with the updated sub-expressions.
  - Return success if the rebuilding succeeds.

ConceptClang: Essential Procedures

Rebuilding Entity References – Extension for Concept Model Checking:

- For each reference to a **DummyAssocDecl**:
  - If the concept model is concrete,
    - Collect valid candidates, and
    - Add the candidates to the model context.
      - Intersecting with pre-existing candidates.
  - If the concept model is an archetype:
    - Simply substitute and rebuild the reference.

Separate Type-checking

PARSING

Generic Component Definition:

```
template<typename InputIterator,
         typename T,
         typename BinaryFunction>
T accumulate( ... ) {
```

Type-Check

Generic Component Use:

```
vector<int> v;
int i = accumulate(v.begin(),
v.end(), 0, plus<int>());
```

Type-Check

INSTANTIATION

Instantiation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op)
{ ... }
```

Type-checking Definitions

PARSING

Generic Component Definition:

```
template<typename InputIterator,
         typename T,
         typename BinaryFunction>
T accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Type-Check

Generic Component Use:

```
vector<int> v;
int i = accumulate(v.begin(),
v.end(), 0, plus<int>());
```

Type-Check

INSTANTIATION

Instantiation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op) {
  ...
}
```

Type-checking Uses

PARSING

Generic Component Definition:

```
template<typename InputIterator,
         typename T,
         typename BinaryFunction>
T accumulate( ... ) { ... }
```

Type-Check

Generic Component Use:

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Type-Check

INSTANTIATION

Instantiation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op) {
  ...
}
```

Type-checking Templates

PARSING

Template Definition:

```
template<typename Iter,
         typename T,
         typename BinOp>
T accumulate(...) { ... }
```

Template Use:

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check

INSTANTIATION

Code Generation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op)
{ ... }
```

Once!

Specialization:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

(Non)Dependent Entity References

- An entity reference: refers to an entity
  - e.g. A **function call expression**: refers to a function declaration
  - e.g. The **use of a type** to declare a variable
- A dependent entity reference:
  - depends on a template parameter
- A non-dependent entity reference:
  - does not depend on a template parameter

EXAMPLE: ACCUMULATE – ALL DEPENDENT ENTITY REFERENCES

```
template<typename InputIterator, typename T, typename BinaryFunction>
T accumulate(InputIterator first, InputIterator last,
             T init, BinaryFunction bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

(Non)Dependent Entity References

- An entity reference: refers to an entity
  - e.g. A **function call expression**: refers to a function declaration
  - e.g. The **use of a type** to declare a variable
- A dependent entity reference:
  - depends on a template parameter
- A non-dependent entity reference:
  - does not depend on a template parameter

EXAMPLE: ACCUMULATE_INT – SOME DEPENDENT ENTITY REFERENCES

```
template<typename Iterator>
int accumulate_int(Iterator first, Iterator last,
                   int init, BinOp<int>& bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

(Non)Dependent Entity References

- An entity reference: refers to an entity
  - e.g. A **function call expression**: refers to a function declaration
  - e.g. The **use of a type** to declare a variable
- A dependent entity reference:
  - depends on a template parameter
- A non-dependent entity reference:
  - does not depend on a template parameter

EXAMPLE: ACCUMULATE_INT – SOME DEPENDENT ENTITY REFERENCES

```
template<template<typename T, typename A = allocator<T> >
         class Container>
int accumulate_int(typename Container<int>::iterator first,
                   typename Container<int>::iterator last,
                   int init, BinOp<int>& bin_op) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Templates: No Separate Type-checking

PARSING

Template Definition:

```
template<... class Container>
int accumulate_int(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Type-check non-dependent

Template Use:

Check arguments

```
vector<int> v;
int i = accumulate_int<vector>
(v.begin(), v.end(), 0, add_int);
```

INSTANTIATION

Code Generation:

```
int accumulate_int(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Type-check dependent

Specialization:

```
int accumulate_int
(vector<int>::iterator first,
 vector<int>::iterator last,
int init, BinOp<int>& bin_op);
```

Type-checking Constrained Templates

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires (InputIterator<I>,
          BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Constrained Template Use:

```
vector<int> v;
int i =
accumulate<vector<int>::iterator,
           int,
           plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Check Constraints-Check

INSTANTIATION

Code Generation:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op)
{ ... }
```

Once!

Specialization **+ Models**:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

Separate Type-checking ?

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires(InputIterator<I>,
         BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Type-check all

Constrained Template Use:

```
vector<int>
int i =
accum
                plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Type-check arguments

Constraints-check arguments

INSTANTIATION

Code Generation:

```
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Specialization + Models:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

Not Quite Separate Type-checking

PARSING

Constrained Template Definition:

```
template<typename I, typename T,
         typename BinOp>
requires (InputIterator<I>,
          BinaryFunction<Op>, ...)
T accumulate(...) { ... }
```

Type-check all

Constrained Template Use:

```
vector<int>
int i =
accum
                plus<int> >
(v.begin(), v.end(), 0,
 plus<int>());
```

Type-check arguments

Constraints-check arguments

INSTANTIATION

Code Generation:

```
int accumulate(...) {
  for (; first != last; ++first)
    init = bin_op(init, *first);
  return init;
}
```

Potential ambiguities

Specialization + Models:

```
int accumulate
(vector<int>::iterator first,
 vector<int>::iterator last,
 int init, plus<int> bin_op);
```

```
InputIterator<vector<int>::ite
rator>,
BinaryFunction<bin_op>, ...
```

Implicit Concepts

✧ Concept Definition:

- Use-patterns
  - To specify requirements
- Axioms
  - Not checked

✧ Concept Model:

- Implicit modeling, only.
  - **== Structural conformance**
- Requirement satisfaction:
  - Find valid expressions

✧ Constrained Template Definition:

- Match expression trees against use-patterns

✧ Constrained Template Use:

- Unchanged

Implicit Concepts: Example

EXAMPLE

```
concept InputIterator<TrivialIterator Iter>
        where EqualityComparable<Iter>    // Refinements
              && Assignable<Iter>
              && Arrow<Iter> {
    Integer difference_type;  // the type of distance between
                              // two input iterators
    Var<Iter> p;
    const Iter::value_type& v = *p;    // converts to
    const Iter::value_type& v2 = *p++; // converts to
};
```

- Reference:
  - G. Dos Reis and B. Stroustrup. *Specifying C++ concepts*. In Proc. 33rd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL), pages 295–308. ACM Press, 2006.

Explicit Concepts

✧ Concept Definition:

- Pseudo-signatures
  - To specify requirements
- Axioms
  - Not Supported

✧ Concept Model (Template):

- Explicit modeling, only.
  - **== Named conformance**
- Requirement satisfaction:
  - Build candidate set

✧ Constrained Template Definition:

- Name-lookup in constraints environment

✧ Constrained Template Use:

- Unchanged

Explicit Concepts: Example

EXAMPLE

```
template<>
concept InputIterator<vector<int>::iterator> {

    // Associated type satisfactions – implicit in this case
    //typedef postincrement_result vector<int>::iterator;

    // Associated function satisfactions
    pointer operator->(vector<int>::iterator) { … }
    vector<int>::iterator& operator++(vector<int>::iterator&) { … }
    postincrement_result operator++(vector<int>::iterator&, int) { … }
    reference operator(const vector<int>::iterator&) { … }
};

// Refinements and associated requirements are
// satisfied by defining models for them.
```

- Reference:
  - Douglas Gregor, Jeremy Siek Douglas, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concepts for c++0x revision 1. Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, august 2005.

Pre-Frankfurt Design

✧ Concept Definition:

- Pseudo-signatures
  - To specify requirements
- Axioms
  - Not checked

✧ Concept Model (Template):

- Implicit and explicit modeling
  - Explicit, by default
  - Implicit under “auto” keyword
- Requirement satisfaction:
  - Build candidate set

✧ Constrained Template Definition:

- Name-lookup in constraints environment

✧ Constrained Template Use:

- Unchanged

➔ Implementation prototype available: ConceptGCC

Pre-Frankfurt Design: Example

EXAMPLE

```
concept_map InputIterator<vector<int>::iterator> {

  // Associated type satisfactions – implicit in this case
  //typedef value_type vector<int>::iterator::value_type;
  // . . .

  // Associated function satisfactions
  pointer operator->(const vector<int>::iterator&) { … }
};

// Refinements and associated requirements are
// satisfied by defining models for them.
```

- References:
  - C++ Standards Committee. *Working Draft, Standard for Programming Language C++. Technical Report N2914=09-0104*, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, June 2009.

Palo Alto Design

✧ Concept Definition:

- Use-Patterns
  - To specify requirements
  - Extended type annotations
- Axioms
  - Not checked

✧ Constrained Template Definition:

- Match expression trees against use-patterns
- TBD

✧ Concept Model:

- Implicit modeling, only.
  - **== Structural conformance**
- Requirement satisfaction:
  - Find valid expressions

✧ Constrained Template Use:

- TBD

➔ Focal points: What are concepts? How to use them?
➔ Language mechanics not addressed yet
➔ Designed for the Standard Template Library

Palo Alto Design: Example

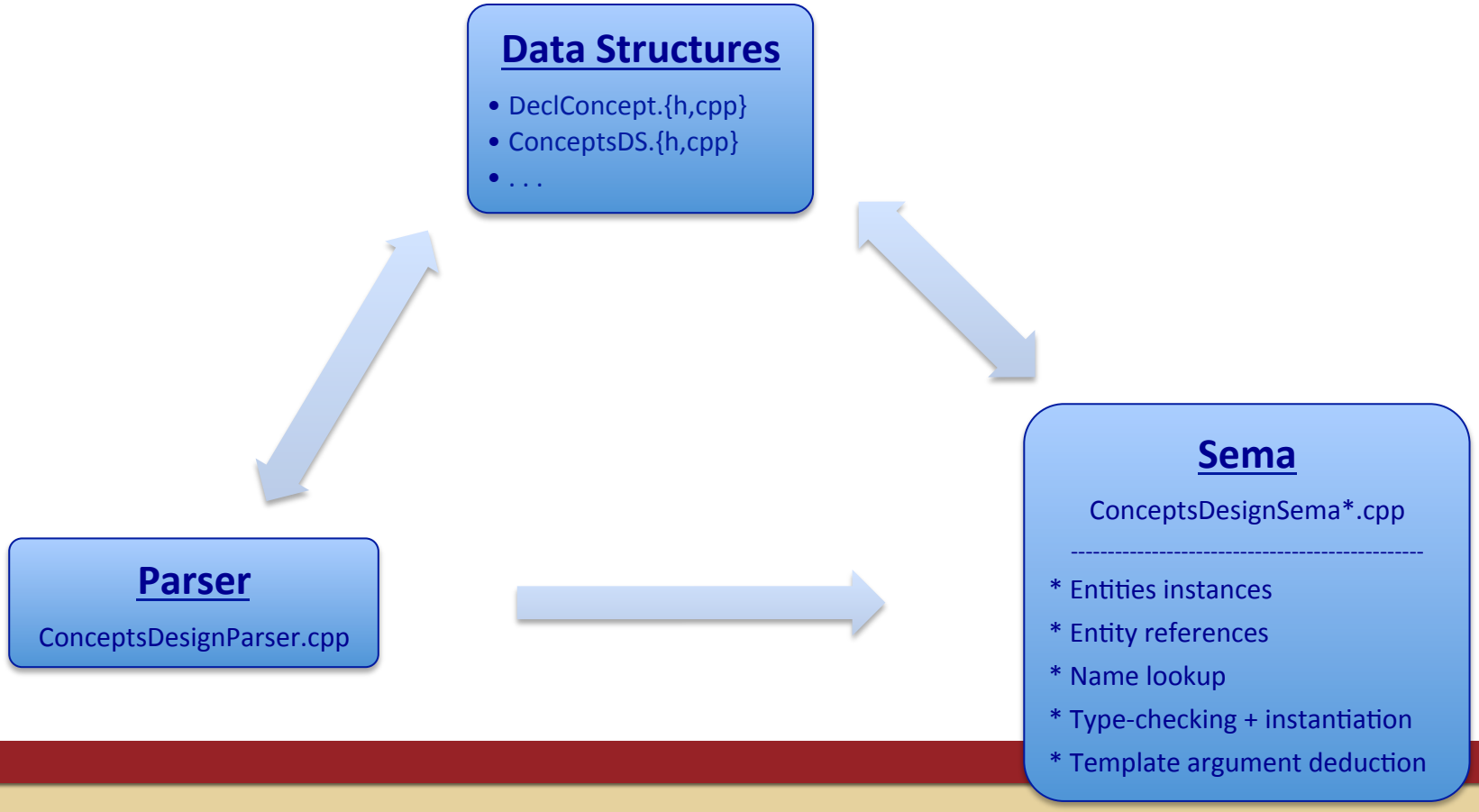
EXAMPLE

```
concept EqualityComparable<typename T> =
     requires (T a, T b, T c) {           // requires clause
          bool { a == b };                    // expression requirements
          bool { a != b };
          axiom { a == b <=> eq(a, b); }  // axiom clauses
          axiom {
               a == a;                        // axioms
               a == b => b == a;
               a == b && b == c => a == c;
          }
          axiom { a != b <=> !(a == b); }
     };
concept WeakInputIterator<WeaklyIncrementable I> =
     Readable<I> &&                       // refinement
     requires (I i) {
          IteratorCategory<I>;            // type requirements
          Derived<IteratorCategory<I>, weak_input_iterator_tag>;
          Readable<decltype(i++)>;
     };
```

- References:
  - C++ Standards Committee. *A Concept Design for the STL. Technical Report N3351=12-0041*, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Jan 2012.

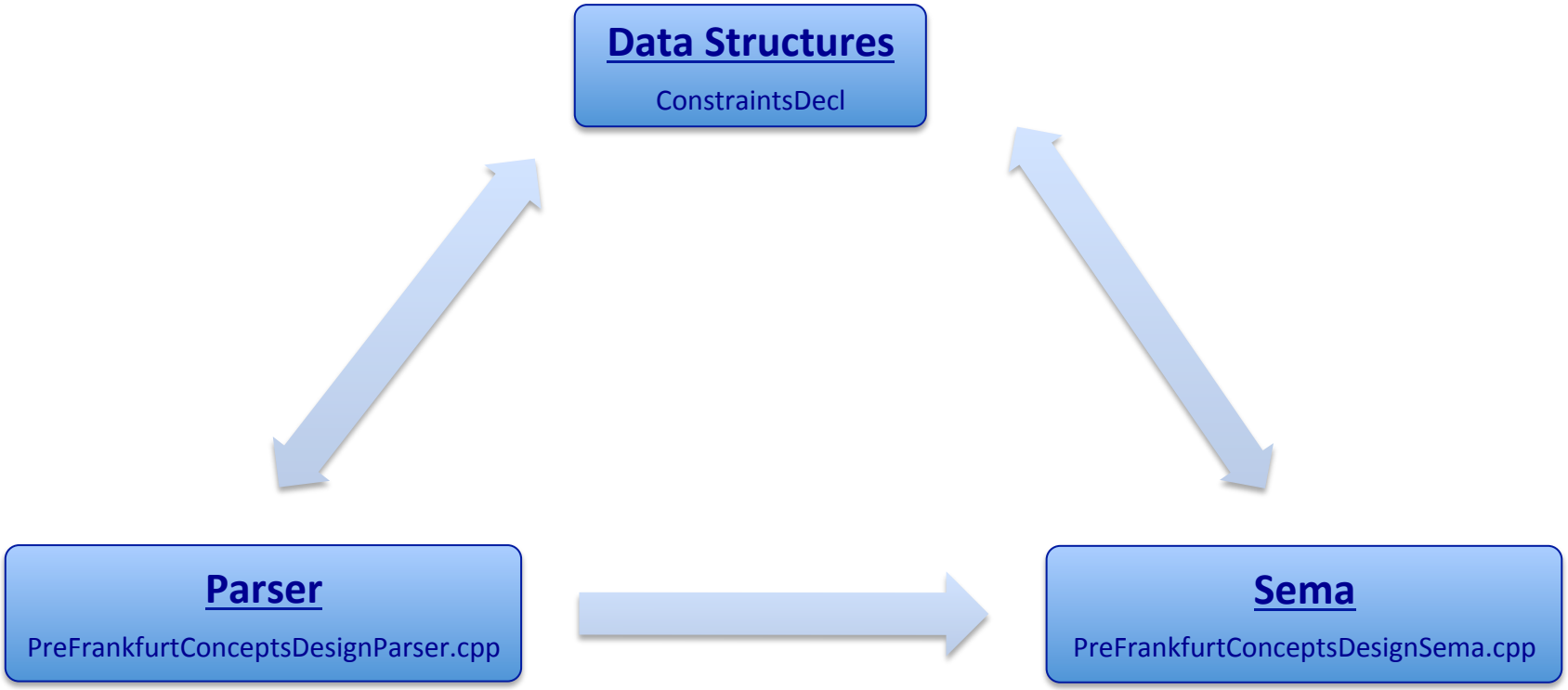
The Infrastructure layer

Interface defined in: Basic/ConceptsDesign.{h,cpp}



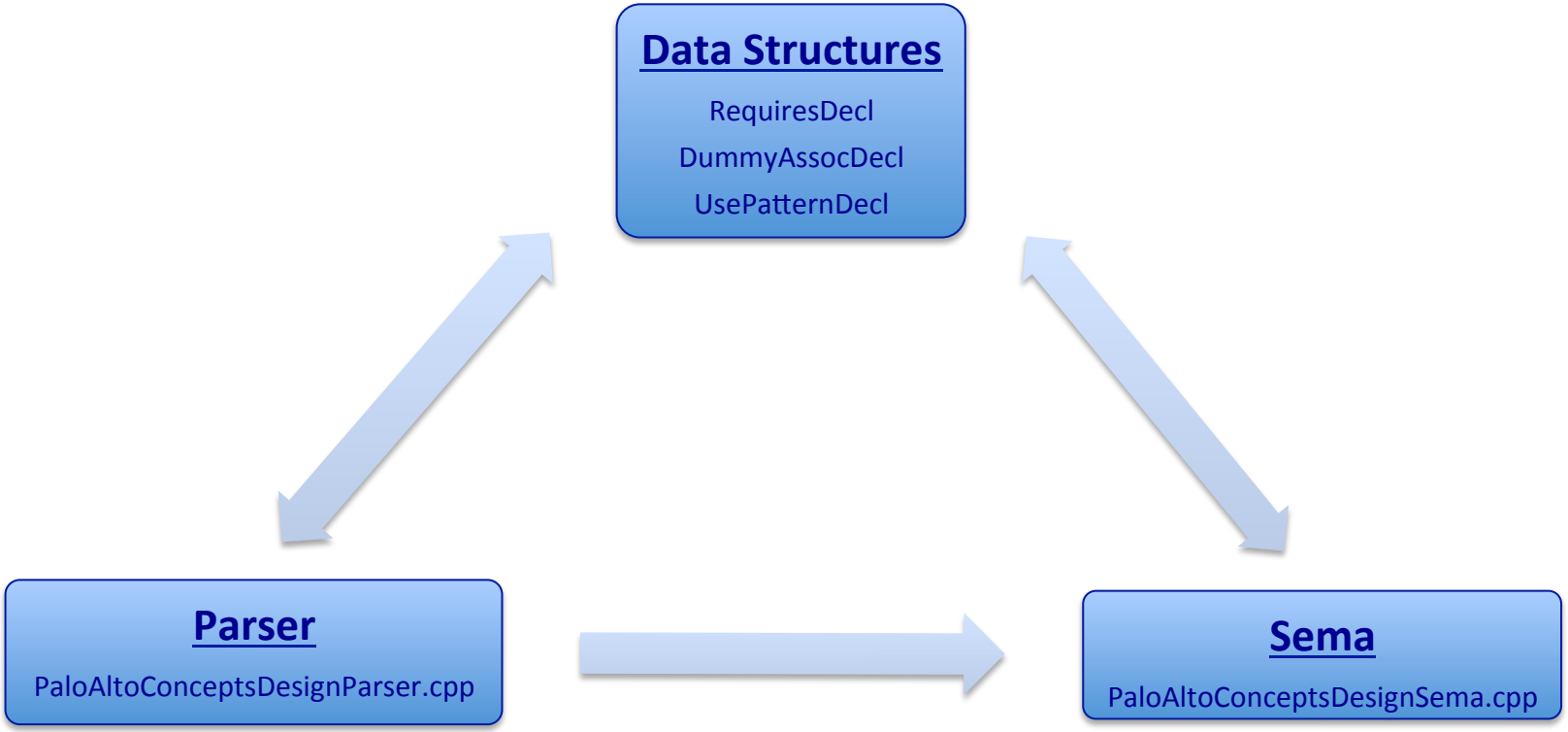
The pre-Frankfurt Instantiation layer

Interface defined in: Basic/PreFrankfurtConceptsDesign.{h,cpp}



The Palo Alto Instantiation layer

Interface defined in: Basic/PaloAltoConceptsDesign.{h,cpp}



Instantiating A Concepts Design

- Start with the interface file:
  - Basic/ConceptsDesign.h
- Define subclasses of **ConceptsDesignParserImpl** and **ConceptsDesignSemaImpl**.
- Define all necessary data structures.
- Inform ConceptClang about the new additions
  - In Basic/ConceptsDesign.cpp, update
    - Sema::CreateUniqueConceptDesignObject()
    - Parser::CreateUniqueConceptDesignObject()
  - Update Basic/PaloAltoConceptsDesignInstances.h
  - Update AST/TraverseConceptsDesignNodes.h
    - If any new node were added to the AST.
- Compile and …. Happy Hacking!

Pre-Frankfurt Instantiation: Summary

- Pseudo-signatures:
  - + Probably complicate the satisfaction of concept requirements, esp since
  - + The satisfaction of associated functions must collect candidates via
    - Generating dummy expressions, name lookup, then overload resolution, or
    - Name lookup, then filtering-out less cv-qualified.
- Explicit and implicit modeling mechanisms:
  - – A complete reuse of ConceptClang infrastructure, mostly.
- No concept overloading:
  - – Reduces the preliminary checking for each concept definition.
- Some subtle extensions:
  - + Allow the shadowing of associated type declarations:
    - Only over associated type declarations of refinements
    - To override default implementations.
  - + Allow the redefinition of pre-defined functions, including builtins:
    - Once, in the scope of concept definitions or models.

Palo Alto Instantiation: Summary

- Use-patterns:
  - + Extend expression parsing mechanism.
    - New expression nodes, parsing contexts, and markers for validation.
  - + Extend name lookup mechanism.
    - Generate dummy associated declarations for concept definitions.
  - – Probably simplify the satisfaction of concept requirements, but
  - + Extend template instantiation mechanism.
    - Replace dummy associated declarations with function candidates
  - + Require an additional procedure: Expression validation.
  - – Eliminate references to associated types.
- Implicit modeling mechanism only:
  - – Eliminates the parsing of models
    - And checking of model templates
  - – Eliminates two areas of reuse of constraints satisfaction
- Concept overloading:
  - + Increases the preliminary checking for each concept definition