Scoping Rules on a Platter

A Framework for Understanding and Specifying Name Binding

Larisse Voufo Marcin Zalewski Andrew Lumsdaine
Center for Research in Extreme Scale Technologies, Indiana University, USA
{lvoufo,zalewski,lums}@crest.iu.edu

Abstract

We present a simple and generic way to reason about *name binding*. Name binding is an essential component of every nontrivial programming language, matching uses of names, *references*, with the things that they name, *declarations*, based on *scoping rules* defined by the language. The definition of name binding is often entangled with the language-specific details, which makes abstract and comparative analysis of competing designs challenging. We present a framework that allows to abstract the fundamental notions of references, declarations, and scopes, and to express scoping rules in terms of four scope combinators and three properties of a specific programming language encapsulated in a concept named `Language`. Using this framework, we clarify complex scoping rules like *argument-dependent lookup* in C++, investigate the implications of the *concepts* feature for C++, and introduce a novel scoping rule named *weak hiding*. In an ideal world, specifications could be formulated based on our framework, and compilers could use such formulation to unambiguously implement name binding. While our examples are primarily centered around C++ and lexical scoping, our framework has applications in other languages and dynamic scoping.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

General Terms Languages, Framework, Binding, Scope, Lookup, Resolution

Keywords Concepts, C++, Name Binding, Name Lookup, Name Resolution, Scoping Rules, Combinators

1. Introduction

In programming languages, *name binding* refers to the process of matching uses of names, *references*, to the things that they name, *declarations*, based on a set of *scoping rules* [1, 12, 32, 34, 41, 42]. This process is essential to every nontrivial programming language, but how a language defines it can be cumbersome to understand and analyze. It is typical for programming languages to define it as part of their (standard) specifications, but such documents can be rather lengthy and

WGP'14, August 31 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2633628.2633633

complex to sort through, especially when it comes to extracting the language-specific definition of name binding. Usually, the definition of name binding is either implicit or tightly intertwined with the definitions of other features. From one language to another, the definition of name binding may even use different terminology; making it difficult to understand exact differences. Similarly, it is difficult to assess how proposed features such as *concepts* [4, 19, 47], *argument-dependent lookup* (in C++ [5]) or *type-directed name resolution* (in Haskell [30]) affect name binding.

To simplify such challenges, we present a framework for understanding and specifying name binding independently of language-specific details. Our framework abstracts the fundamental notions of *references*, *declarations* and *scopes*, decoupling the expression of scoping rules from their execution. For different languages, features, or even kinds of reference, we express scoping rules using a minimal set of four scope combinators that we have defined, plus three properties of name binding that we have identified. We encapsulate the said properties in a concept named `Language`, instances of which allow to express salient differences in scoping rules between different languages.

We developed this framework in an effort to understand how a proposed design for concepts would affect C++ and to propose an extension to name binding that we found necessary for concepts. This extension turned out to be a new

scoping rule, called *weak hiding*, with applications in other languages that support any form of constraints-based polymorphism likened to concepts, like Haskell. The framework allowed us to not only describe our extension, but to explore how it could be integrated in current compilers. In particular, we ended up with a novel mechanism for name binding called *two-stage name binding* (*Bind*$^{\times 2}$) that essentially consists of iteratively applying current mechanisms for name binding, under different contexts.

The rest of this chapter is structured as follows. In the next section, we revisit our problem statement, describing the issues that this framework resolves in greater detail. Then, in Sect. 3, we introduce our framework, with its generic definition of name binding. In Sect. 4, we cover applications of the framework which range from expressing complex features like argument-dependent lookup (Sect. 4.1) and uses of operators in C++ (Sect. 4.2), to expressing concrete differences between various languages, features, or even kinds of references (Sect. 4.3), to exploring compiler integrations and to concretely motivating our proposed *weak hiding* and *Bind*$^{\times 2}$ (Sect. 4.4). A discussion on background, ongoing and related work follows in Sect. 5, and we conclude in Sect. 6. An implementation of our framework is available online [6].

2. Problem Statement

Consider a call to some function named `move`, cf. Fig. 1 for example, and attempt to explain how the call is type-checked in three different languages: C++ [5, 51], Haskell [30], and Common Lisp [45]. Assuming basic understanding of programming languages theory, the explanation quickly follows drastically different paths, including different choices of terminology. On the one hand, the C++ path requires understanding features like templates, concepts, (non) dependent names, overload resolution, namespaces, `using` declarations, and at some level, different compilers and their interpretations of the specification, e.g., GCC and Clang. On the other hand, the Haskell path requires understanding the Hindley-Milner type system, type classes, type inference, dependency analysis, dictionary-passing style, modules, `import` declarations, and at some level, also different compilers, e.g., GHC, Hugs and NHC. On the third hand, the Common Lisp path requires, among others, understanding dynamic scoping as enabled via the `defvar` and `declare` constructs.

Programming languages typically define the features that they support in a specification document, which includes a definition of name binding as well as the scoping rules to apply for various kinds of references. For example, a specification document may dictate how to bind uses of types, functions, or record field names; and may do so differently for each case. A recurring issue with the specifications, however, is that they tend to be lengthy and too complex to understand. For example, in C++, the definition of name binding spans through about four chapters of the specification document and contains links to various other parts of the docu-

ment. In Haskell, while the specification document may not be as lengthy, the sections that cover name binding also contain links to other parts of the document. We can only anticipate similar observations within other languages, depending on their practical use.

The difficulties with extracting the meaning of name binding from language specifications increases the challenges in understanding and expressing the binding of various names. In fact, we have noticed that production grade C++ compilers like Clang [8], GCC [13], the Intel compiler [22], and Microsoft Visual C++ [31] implement uses of operators and *argument-dependent lookup* (ADL) either incorrectly or inconsistently (Sects. 4.1 and 4.2).

Adding new features to a language makes understanding name binding even more challenging, since the addition may affect the meaning of name binding in unanticipated ways. For example, ADL in C++ is intended to facilitate uses of operators defined in different namespaces. However, after several revisions of the standard, it has become a classic example of a feature with very complex implications. As of today, for nearly every new proposed feature, compatibility with ADL has become a standard concern [2, 3, 16, 49]. Moreover, even uses of operators have gained complexity with ADL; and that is in addition to the fact that they follow slightly different scoping rules than function calls.

2.1 Towards the Name Binding Framework

We started looking at specifications for name binding in an effort to help design concepts for C++ using ConceptClang [53, 54]. Essentially, we were investigating how a function call like `move()` should be type-checked in a constrained

template. Fig. 1b illustrates the exact problem that we were facing. It turns out that when C++ migrates into ConceptC++ (i.e. C++ with concepts), valid programs such as in Fig. 1a

would no longer be considered valid. This is because the `requires` clause in Line 3 of Fig. 1b restricts the binding of `move` in Lines 8–9 in ways that invalidates the call in Line 9. Namely, without constraints (Fig. 1a, Line 9), concepts are implicit; and with such, all function calls in C++ templates look up declarations in external scopes and are bound via overload resolution—successfully in this example. In contrast, with explicit concepts (Fig. 1b), declarations in outer scopes are hidden by constraints (Line 3), and the function call in Line 9 fails.[1]

Maintaining the backward compatibility of programs as new features are added in a language is very important in software production, and of particular importance to C++. So, Line 9 of Fig. 1b illustrates a problem that was warrant a solution. We actually came up with a solution, but explaining the problem and describing our solution in concrete terms turned out to be challenging and easily prone to bikeshedding into other language details. Thus, we developed the framework that is the essence of this chapter (and introduced in Sect. 3). Using our framework, we are now able to formulate our ideas as follows.

$$\mathit{rotate} \lhd P \lhd T \lhd (N \Leftrightarrow U), \tag{1}$$

$$\textit{rotate} \lhd P \lhd (C_T \Leftrightarrow T) \lhd (N \Leftrightarrow U), \text{and} \quad (2)$$

$$\textit{rotate} \lhd P \lhd (C_T \Leftrightarrow T) \Lleftarrow (N \Leftrightarrow U), \quad (3)$$

Eqs. 1 and 2 describe the issues at hand, i.e., with Line 9 of Figs. 1a and 1b, respectively; while Eq. 3 describes our proposed solution. *rotate* represents the block scope of the function template `rotate`; C_T holds all declarations corresponding to the constraint `C<T>`; P, T, and N, respectively, represent the scopes of function parameters, template parameters, and surrounding namespaces (possibly more than one, assuming that the example represents only a fragment of a complete program); and U represents the scopes imported by declarations in the surrounding namespace.

The problem that we had (Fig. 1b), where uses of outer-scope declarations are invalidated by explicit constraints, was essentially a scoping rules problem; and to resolve it, we have identified four elementary scoping rules that we named *hiding*, *merging*, *opening* and *weak hiding*, and respectively denoted with symbols $\lhd$, $\Leftrightarrow$, $\rhd$ and $\Lleftarrow$. The first two rules, *hiding* ($\lhd$) and *merging* ($\Leftrightarrow$), correspond to the most common forms of scoping rules. Given two scopes A and B, $A \lhd B$ corresponds to searching for declarations in B only if no match is found in A, and $A \Leftrightarrow B$ corresponds to searching for declarations in both A and B as if the scopes were merged into one. We introduce *opening* ($\rhd$) as necessary for expressing ADL in C++, and propose *weak hiding* ($\Lleftarrow$) as a new scoping rule for checking name uses in constrained environments—or *restricted scopes*. One may think of *opening* as a dual of *hiding*, that has $A \rhd B$ correspond to searching for declarations in B when a match is found in A; and of

weak hiding as a sweet middle between *hiding* and *merging*, that has $A \Leftarrow\!\!\mid B$ correspond to searching for declarations in B only if no viable match is found in A.

To express the subtle distinctions between all four scoping rules, we had to take a closer look at the process of name

[1] Our analysis is based on a proposal for C++ concepts dubbed *pre-Frankfurt* [5, 20]. In a recent proposal dubbed *Palo Alto* [47], the body of constrained templates is checked using a procedure called *expression validation*, which has not yet been specified. When specified, such a procedure can only lead to a program behavior isomorphic to that of the pre-Frankfurt design. Thus, we consider the descriptions herein provided as *implicitly* applying to the Palo Alto alternative.

```
template<typename I>
// implicitly requires RandomAccessIterator<I>
// and Permutable<I>, with move(ValueType<I>&&)
I rotate (I f, I m, I l) { ...
  I p = f;
  ...
  if (__is_pod(ValueType<I>) ...) {
    ValueType<I> t = move(*p);
    move(p+1, p+l-f, p);       // OK.
    ...
  }
  ...
}
```

(a) With implicit concepts in C++

```
template<typename I>
requires RandomAccessIterator<I> &&
      Permutable<I> // => move(ValueType<I>&&)
I rotate (I f, I m, I l) { ...
  I p = f;
  ...
  if (__is_pod(ValueType<I>) ...) {
    ValueType<I> t = move(*p);
    move(p+1, p+l-f, p);       // Not OK?
    ...
  }
  ...
}
```

(b) With explicit concepts in ConceptC++

Figure 1: Name lookup and resolution with and without concepts

This program is adapted from the latest release (4.6) of the GNU C++ Library [15]. It is a fragment of a specialization of the `rotate()` algorithm, from the standard template library [4, 33, 46], for random access iterators [14]. We have removed the `std::` qualifiers for simplicity. One-argument calls to `move`, such as in Line 8 are supposed to be covered by the constraint in Line 3, while three-argument calls to `move` are bound to an outer scope declaration that is properly constrained.

binding itself as it executes each one of the scoping rules; our goal being to break it down into appropriate stages. We ended up with a view of name binding as a composition of *name lookup* and *name resolution*, where name lookup gathers all declarations matching a given reference, while name resolution reduces the matches down to the best viable declaration. With this compositional view, we note that usually, only name lookup determines which declarations to hide or merge. But, it is only with weak hiding that name resolution also determines which declarations to hide (because the check whether a matching declaration is actually viable for a reference happens during name resolution). Without weak hiding, a non-viable declaration may hide a viable declaration; while with weak hiding, only viable declarations get to hide other declarations.

Indeed, our proposed solution (Eq. 3) uses weak hiding; and with that, the program in Fig. 1b is no longer invalid because name binding does not halt when it finds the constraints and fails, but rather moves on with looking up names in outer scopes, where it finds a viable match for the call in Line 9.

2.2 Two Layers of Abstraction

Our framework defines the denotations $\lhd$, $\Leftrightarrow$, $\rhd$ and $\Lhd$ as combinators acting on scopes—or *scope combinators*, based on a compositional view of name binding as name lookup followed by name resolution. This said, Eqs. 1-3 are *scope expressions* that can be executed by compilers [6]. How the scope

expressions behave depends on properties of the language that can be expressed in either one of two ways. On the one hand, one may

- express the scoping rules using the scope combinators. On the other hand, one may
- express other properties not captured by the combinators in terms of three salient properties, which are:
  - how a language determines the potential matches,
  - how it checks that a declaration is viable, and
  - how it handles ambiguous matches (which may not always result in a failure).

The first two properties specify how names are hidden, while the second one indicates support for overloading (a.k.a. *ad-hoc* polymorphism that is not based on type-class constraints), viability checking, or related variants. Based on these properties, we can express other points of differences in name binding beyond how the scopes are combined. In

particular, we can clarify aspects of proposed features like multimethods [35] for C++ or TDNR for Haskell; providing a different way to explain how TDNR is not overloading. We can also explore possible effects of proposed features like weak hiding on languages like Haskell. Sect. 4.3 covers these examples.

For generality purposes, we have encapsulated the above

salient properties in a concept named `Language`.To use our framework, one must instantiate this concept for different references, declarations, features or languages.

Naturally, the `Language` concept, as a generic tool, brings in a range of possibilities for additional abstractions and experimentations. In particular, it has allowed us to investigate different ways in which one could integrate our combinators in current compilers [6]. A notable outcome of this investigation is a novel mechanism for name binding, implementing weak hiding, that we call *two-stage name binding* (*Bind*$^{\times 2}$). *Bind*$^{\times 2}$ is designed to facilitate the integration of weak hiding in current compilers and has been implemented for C++ concepts [53]. We briefly introduce it in Sect. 4.4, and a more thorough definition and description of *Bind*$^{\times 2}$ falls outside the scope of this paper.

3. The Name Binding Framework

We begin with a general definition of name binding. As stated in Sect. 1, it is a process that matches a *reference* to a *declaration*, if possible. As such, it takes in a representation of a reference, **Ref**, including any contextual information about the reference, and returns either a declaration, **Decl**, or an error, **Error**. One essential contextual information of the reference is a scope, **Scope**, which we single out from all the other information for genericity purposes. That said, we define name binding as a function

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

such that

$$\textbf{\textit{bind}}\ \ (\mathit{ref}, \mathit{scope}) = ((\textbf{\textit{resolve}}\ \mathit{ref}) \circ (\textbf{\textit{lookup}}\ \mathit{ref}))\ \mathit{scope};$$

where ***lookup*** is defined as a function from a reference and a scope to an overload set; and ***resolve*** is defined as a function from the overload set to, potentially, a declaration. We represent scopes as mappings from references to sets of matching declarations. Thus, given a reference *ref*, ***lookup*** queries a scope for its matching declarations, a.k.a. an *overload set*. In other words, using Haskell as specification language, we could define a scope (**Scope**) as a data structure

```
data Scope ref decl =
 Scope { lookup :: ref → OverloadSet ref decl }
```

in which ***lookup*** (`lookup`) is considered a property of scopes, and an overload set (`OverloadSet`) is defined as a data structure

```
data OverloadSet ref decl = OverloadSet {
    result :: Set decl
  , null :: Bool
  , resolve :: ref → Maybe decl
  , merge :: OverloadSet ref decl →
      OverloadSet ref decl
  }
```

where `result`, `null`, `resolve`, and `merge` are properties of overload sets, respectively representing

- the result of name lookup,
- whether name lookup found any matching declaration,
- how to resolve the lookup results down to a single declaration, and
- how to merge a given overload set with another.

In this definition, **Scope**s are considered entities that can be combined into other **Scope**s based on some scoping rules defined by the language. In the next sections (Sects. 3.1 and 3.2), we define four (4) scope combinators: three (3) based on scoping rules as we currently know them, i.e., *hiding* ($\lhd$), *merging* ($\Leftrightarrow$) and *opening* ($\rhd$), and one based on our novel scoping rule, *weak hiding* ($\Lhd$).

3.1 The Scope Combinators, Currently

Consider the function calls in Lines 4, 13 and 25 of the C++ program in Fig. 2. Respectively, they illustrate the *hiding*, *merging* and *opening* scoping rules, which we define as scope combinators

$$\begin{aligned}
\lhd\ ,\ \Leftrightarrow\ ,\ \rhd\ &: \mathbf{Scope}_{\mathrm{Ref,Decl}} \times \mathbf{Scope}_{\mathrm{Ref,Decl}} \to \mathbf{Scope}_{\mathrm{Ref,Decl}} \\
\mathbf{s_1} \Leftrightarrow \mathbf{s_2} &= \boldsymbol{lookup}_{ref}\ \mathbf{s_1} \cup \boldsymbol{lookup}_{ref}\ \mathbf{s_2} \\
\mathbf{s_1} \lhd \mathbf{s_2} &= \boldsymbol{lookup}_{ref}\ \mathbf{s_1}\ \mathbf{?}\ \boldsymbol{lookup}_{ref}\ \mathbf{s_1}\ \mathbf{:}\ \boldsymbol{lookup}_{ref}\ \mathbf{s_2} \\
\mathbf{s_1} \rhd \mathbf{s_2} &= \boldsymbol{lookup}_{ref}\ \mathbf{s_1}\ \mathbf{?}\ \boldsymbol{lookup}_{ref}\ \mathbf{s_2}\ \mathbf{:}\ \mathbf{empty}
\end{aligned}$$

using C++'s conditional operator, $\mathtt{?}\,\mathbf{:}$, for simplicity.

By the general definition of combinators, each scope combinator, i.e. $\Leftrightarrow$, $\lhd$, and $\rhd$, takes two scopes and returns a new scope. Given a reference *ref*, *merging* ($\Leftrightarrow$) joins the results of name lookup in both scopes, while *hiding* ($\lhd$) and *opening* ($\rhd$) decide whether to perform name lookup in the second scope, $\mathbf{s_2}$, based on whether name lookup in the first scope, $\mathbf{s_1}$, is successful. If lookup is successful in $\mathbf{s_1}$, then *hiding* considers only the first scope, ignoring the second scope. In contrast, *opening* considers only the second scope, ignoring the first one. If lookup is not successful in $\mathbf{s_1}$, then *hiding* considers the second scope; while *opening* considers the first one, which happens to be empty. In the above definition, we write **empty** solely for clarification purposes.

Using these combinators, one may express the scoping rules for each call, respectively, as

$$\mathbf{test} \lhd \mathbf{::}\ , \tag{4}$$

$$(\mathbf{test} \Leftrightarrow \mathbf{ns}) \lhd \mathbf{::}\ , \quad \text{and} \tag{5}$$

$$(\mathbf{test} \Leftrightarrow \mathbf{ns} \Leftrightarrow (\mathbf{ns} \rhd (\mathbf{test} \lhd \mathbf{adl}))) \lhd (\mathbf{::} \Leftrightarrow \mathbf{adl})\ , \tag{6}$$

where **test** represents the block scope of the functions `test_hiding`, `test_merging` and `test_opening`, **ns** and **adl** represent separate namespace scopes, and **::** represents the outer (namespace) scope.

```
1  void foo();
2
3  void test_hiding() {
4    foo();
5  }
6
7  namespace ns {
8    void foo(int);
9  }
10
11 void test_merging() {
12   using ns::foo;
13   foo();
14 }
15
16 namespace adl {
17   struct X {};
18   void foo(typ);
19  }
20
21 void test_opening() {
22   adl::X x;
23   using ns::foo;
24   // void foo();
25   foo(x);
26 }
```

The *hiding* and *merging* rules are the most common in programming languages. While *hiding* corresponds to name shadowing, *merging* corresponds to the union of different scopes which we typically observe with features like mixins, module imports, or C++'s `using` declarations. For example, for the first call (Line 4, Eq. 4), ***lookup*** does not find anything in **test** and thus proceeds to the outer scope where it finds the declaration in Line 1. Later, ***resolve*** finds the found declaration (Line 1) viable for the call, and the call succeeds. In contrast, for the second call (Line 13, Eq. 5), ***lookup*** finds the declaration in Line 8, which ***resolve*** does not find viable;

Figure 2: Using *hiding*, *merging* and *opening* in C++

and the call fails. ***lookup*** does not consider the outer scope because the found declaration (Line 8) has been merged into **test** via the `using` declaration in Line 12, and thus hides the declaration in the outer scope.

The *opening* rule is, interestingly, not trivially observed. In fact, we found it necessary to define only so that we can describe ADL in C++ (Sect. 4.1). One may view it as a dual of *hiding* since its behavior, under the same condition, is opposite to that of *hiding*. For example, consider the third call (Line 25, Eq. 6) in Fig. 2. ADL dictates that, if a call argument (e.g., `x`) is such that its type (e.g., `X`) is defined in a given namespace (e.g., `adl`), then that namespace should also be taken into consideration by name lookup; but that is only if there is a matching declaration and no matching block scope declaration. As a result, ***lookup*** finds and merges all declarations from scopes **test**, **ns** and **adl**; and since the declaration in Line 18 is the only one viable for the call, the call succeeds; but that is only because Line 24 is commented out. Uncommenting Line 24 causes the declaration in **test** to hide declarations in **adl**, as well as those in the outer scope. In other words, uncommenting Line 24 disables ADL if there is a match in **ns**, which causes the call to fail.

To simplify expressions that would use these combinators – i.e. *scope expressions*, as well as to reflect general practices, we make the combinators right-associative, and give *merging* a higher precedence than the other combinators. So, an expression $\mathbf{s_1} \lhd \mathbf{s_2} \Leftrightarrow \mathbf{s_3} \lhd \mathbf{s_4}$ is equivalent to $\mathbf{s_1} \lhd ((\mathbf{s_2} \Leftrightarrow \mathbf{s_3}) \lhd \mathbf{s_4})$.

Note that both *hiding* and *opening* are actually special cases of a more general (ternary) combinator that one could define

as follows (with $\mathbf{Scope}_{\mathrm{RD}} = \mathbf{Scope}_{\mathrm{Ref,Decl}}$):

$$\begin{aligned} &\texttt{<?>} : \mathbf{Scope}_{\mathrm{RD}} \times \mathbf{Scope}_{\mathrm{RD}} \times \mathbf{Scope}_{\mathrm{RD}} \to \mathbf{Scope}_{\mathrm{RD}} \\ &\mathbf{s_0} \texttt{ <?> } \mathbf{s_1}\ \mathbf{s_2} = \mathit{lookup}_{\mathit{ref}}\ \mathbf{s_0}\ \mathbf{?}\ \mathit{lookup}_{\mathit{ref}}\ \mathbf{s_1}\ \mathbf{:}\ \mathit{lookup}_{\mathit{ref}}\ \mathbf{s_2}\ . \end{aligned}$$

For simplicity and practical reasons, we do not focus on this generalized definition in this paper.

3.2 *Weak Hiding*, a New Scoping Rule

With the current rules, only name lookup, and never name resolution, participates in the decision of which scope to process. As a result, as illustrated in Fig. 1b, programs are considered invalid which, at least for practical purposes, should

```
1  concept SweetAddable<Semiregular T> =
2    requires (const T& a, T&& b) {
3      T&& = a + b;
4      T&& = b + a;
5    };
6
7  template<typename T>
8    requires ... std::SweetAddable<T> ...
9  void test(T&& a, T&& b) {
10   T s1;
11   T s2;
12   T s3 = s1 + T() + T() + s2; // Not OK?
13 }
```

Figure 3: Ambiguity Resolution in ConceptC++

not be. To resolve this issue, we introduce *weak hiding*, to allow name resolution to participate in the decision as well. This said, we define it as scope combinator

$$\Leftarrow\!\!\!\!\mid \;:\; \mathbf{Scope}_{\mathrm{Ref,Decl}} \times \mathbf{Scope}_{\mathrm{Ref,Decl}} \to \mathbf{Scope}_{\mathrm{Ref,Decl}}$$

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

with the same precedence and associativity properties as established on the earlier combinators.

One can view *weak hiding* as a sweet middle between *hiding* and *merging* where a scope is processed conditionally on whether name resolution over the result of name lookup is successful. For example, consider the second call (Line 13, Eq. 5) in Fig. 2. If the relationship between **test** and **::** were

weak hiding instead of *hiding*, i.e., with the scoping rules expressed as

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

then ***lookup*** would consider the outer scope after ***resolve*** fails to find the declaration in Line 8 viable for the call. Then, as with the first call (Line 4, Eq. 4), it would find the declaration in Line 1 viable; and the call would succeed.

Similarly to the *hiding* scoping rule, we could generalize the definition of *weak hiding* into a ternary combinator; thereby allowing one to extend the reasoning for *opening* into a "*weak opening*" scoping rule. But, for the same practical purposes, let us not venture that way just yet.

Parameterized weak hiding: The above definition of *weak hiding* uses the inherent properties of name resolution in a given language to determine which scope to process. Depending on the intended use of *weak hiding*, this may lead to undesirable program behavior. Consider, for instance, resolving ambiguities in an overload set. In general, if an overload set contains more than one best viable declarations, then this is considered an ambiguity and name resolution fails, which results in an error. For example, take the program in Fig. 3, which illustrates an issue that has been instrumental in making a number of design decisions for C++ concepts [18][2]. This example uses C++11's *lvalue* and *rvalue* references, investigating how move semantics affects concepts and vice-versa.

For the function call in Line 12, there are at least two possible matches in the constraints (Line 8). The function call itself has two *rvalue* arguments, but the matching declarations expect one *lvalue* argument and one *rvalue* argument, each in different orders. With the current meaning of name resolu-

tion, after ***lookup*** finds the matches in the constraints, ***resolve***

[2] The original issue was expressed using pre-Frankfurt *pre-Frankfurt* [5, 20] design syntax. This example re-expresses the issue using the *Palo Alto* [47] design syntax

finds them all to be best and equally viable for the function call; and thus fails. Using the scope expression in Eq. 3 (with *rotate* replaced with **test**), when name resolution fails given the matches in the constraints, *weak hiding* prompts ***lookup*** to process the outer scope; at which point at least two undesirable things may happen. On the one hand, the function call may eventually fail from failure to find a single best viable declaration. On the other hand, it may actually succeed, but with binding to the wrong declaration.

In practice, one may not want name resolution to fail, given the matches in the constraints, if there is a guarantee that the instantiation of the function template `test` will be successful. Instead, one may want to take advantage of the full expressive power of concepts without liability, and allow *weak hiding* to change the meaning of ambiguity if the scope being processed is restricted. In particular, for the program in Fig. 3, it may be reasonable to define *weak hiding* in such a way that whenever the scope being processed is restricted, ambiguities in overload sets are not considered to be errors; but rather result in some representative declaration or "seed". With such definition of *weak hiding*, the call in Line 12 succeeds and the program in Fig. 3 is considered valid.

In more general terms, we would not want to sacrifice expressiveness for safety if we can avoid it; and to this end, we generalize the earlier definition of the *weak hiding* scope combinator as follows

$$\Leftarrow\!\!\shortmid_{\mathcal{U}} : \mathbf{Scope}_{\mathrm{Ref,Decl}} \times \mathbf{Scope}_{\mathrm{Ref,Decl}} \to \mathbf{Scope}_{\mathrm{Ref,Decl}}$$

$$\mathbf{s_1} \Leftarrow\!\!\shortmid_{\mathcal{U}} \mathbf{s_2} = \Big(\boldsymbol{resolve}_{ref} \circ \Big(\boldsymbol{update} \circ \boldsymbol{lookup}_{ref}\Big)\Big)\ \mathbf{s_1}\ ?\ \Big(\boldsymbol{update} \circ \boldsymbol{lookup}_{ref}\Big)\ \mathbf{s_1}\ :\ \boldsymbol{lookup}_{ref}\ \mathbf{s_2}$$

where the subscript $\mathcal{U}$ indicates temporary changes in the properties of name binding—or *bind environment*, such as changes in the meaning of ambiguity, encapsulated in the function **update**. Note that **update** is identity if there is no change in the bind environment.

$\mathcal{U}$ is a property of weak hiding that essentially acts as an additional parameter in this generalized definition of *weak hiding*. Therefore, we refer to the definition as *parameterized* by the bind environment. Indeed, *Parameterized weak hiding* allows properties of name resolution, including the meaning of ambiguity, to change based on the scope.

3.3 The `Language` Concept

The combinators in Sects. 3.1 and 3.2 are not always sufficient to express differences in meanings of name binding. Another set of differences comes from observing the process of name binding itself, given a (combination of) scope(s). Through experimentation, we have observed a recurring pattern which allows to define ***lookup*** and ***resolve*** generically, as follows

```
lookup :: Language r d ⇒ r → Set d → OvSet d
lookup ref decls = Set.filter (match ref) decls

resolve::Language r d ⇒ r → OvSet d → Maybe d
resolve ref decls = assess $
                    select_best_viable ref decls

assess :: Ambiguity d ⇒ BVSet d → Maybe d
assess decls = case (Set.elems decls) of
                 []     → Nothing
                 [decl] → Just decl
                 _      → ambiguity decls
```

where `lookup` and `resolve` respectively represent ***lookup*** and ***resolve,*** and scopes are represented as sets of declarations (`Set`).

In this recurring pattern, name lookup, given an encoding of how to test whether a given reference matches a given declaration, e.g., `match`, simply filters matching declarations out of a given scope. Name resolution, on the other hand, consists of a combination of:

- selecting viable candidates out of an overload set, e.g., with `select_best_viable`, and
- assessing the viable set based on the meaning of ambiguity, e.g., with `assess`.

`select_best_viable` reduces an overload set (`OvSet`) down to only those declarations that are viable for a given reference (`BVSet`); and `assess` reduces the viable declarations down to either a single best viable declaration or an error (`Maybe`). Usually, a single or no viable declaration automatically means success or failure of name binding, while the meaning of more than one viable declarations depends on that of ambiguity (`ambiguity`).

Ambiguity does not have a concrete meaning at such a generic level, since it varies based on the language, features, and perhaps even the scoping rule in use (cf. Sect. 3.2). So, we abstract it into a concept named `Ambiguity`, that is used to define `assess`.

```
1 class Ambiguity d where
2   ambiguity :: BVSet d → Maybe d
3   ambiguity _ = Nothing
```

In general, ambiguity is considered to be an error. So, we define it as such by default. (Instances of `Ambiguity` may override the default behavior.)

Both `select_best_viable` and `match` also do not have concrete meanings at this generic level. `select_best-`

`_viable` usually indicates overloading in the sense of *ad-hoc* polymorphism (not based on type-class constraints) or variants. For example, in Haskell – which does not support *ad-hoc* polymorphism, it may correspond to either identity or type inference depending on how one defines name binding; and each way implies a significant property for the language. Meanwhile, `match` usually indicates various ways of matching references to declarations by name, including any necessary normalization. For example, in C++, function calls and uses of types both match references to declarations the same way, but the normalization for uses of types (i.e. after looking up names of types) may perform ambiguity resolution which that for function calls does not (i.e. after looking up names of functions). `match` also highlights a filtering stage that TDNR adds to Haskell's name lookup without changing its meaning for `select_best_viable`. Sect. 4.3 discusses different interpretations and related implications such as herein highlighted.

Since both `select_best_viable` and `match` act on pairs of references and declarations as defined by the language, we encapsulate them in a concept named `Language`.

```
1 class Ambiguity d ⇒ Language r d where
2   match :: r → d → Bool
3   select_best_viable :: r → OvSet d → BVSet d
```

All three functions `match`, `select_best_viable` and `ambiguity` encapsulate salient properties of elementary name binding as defined by a language; and we reflect that in the `Language` concept, which incidentally refines the `Ambiguity` concept in order to include the ambiguity property into the mix. Aside from these three functions, all other aspects of elementary name binding have meaning at the

generic level and constitute the recurring pattern that we observed.

The `Language` Concept, Extended: In the above definition of elementary name binding, the meaning of name resolution is constant throughout all scopes; and is thus only useful for applications of non-parameterized weak hiding (Sect. 3.2). To take advantage of parameterized weak hiding, name resolution must take into account potential changes in the bind environment. For example, if changes in ambiguity were the only changes that we were interested in expressing (as is the case for the problem in Fig. 3), then one may redefine generic ***resolve*** as

```
1  resolve :: Basic.Language r d ⇒
2             BindEnv d → r → OvSet d → Maybe d
3  resolve env ref decls = assess env $
4                          select_best_viable ref decls
5
6  assess :: BindEnv d → BVSet d → Maybe d
7  assess env decls = case (Set.elems decls) of
8                          []     → Nothing
9                          [decl] → Just decl
10                         _      → ambiguity env decls
```

where `Basic.Language` represents the above definition of the `Language` concept (for non-parameterized weak hiding); and `BindEnv` represents the bind environment and is

defined as follows.

```
1 data BindEnv d = BindEnv
2              { ambiguity :: BVSet d → Maybe d }
```

In this updated definition, name resolution – and more specifically the assessment of viable declarations – now gets the meaning of ambiguity from the bind environment, instead of the language as previously. Indeed, the new definition of `assess` is no longer constrained by the `Ambiguity` concept, but rather now takes in an additional input, `env`, which represents the bind environment.

Given this update, we can specify whether and how the bind environment changes, by encapsulating the changing behavior in a concept named `Parameterized` that we then extend our `Language` concept with. Thus, we define an extended form of the `Language` concept as

```
1 class (Parameterized d, Basic.Language r d) ⇒
2       Language r d where
```

which refines the `Parameterized` concept, defined as

```
1 class Parameterized d where
2   get_env :: Maybe (BindEnv decl)
3   get_env = Nothing
```

where `get_env` represents the changing behavior. By default, we assume no change in the bind environment, and reflect that in the definition of `Parameterized`. When the meaning of ambiguity changes, say to treat ambiguities as non-errors (`amb_is_not_err`), one may express that by in-

stantiating the `Parameterized` concept as follows.

```
1 instance Parameterized (WeakHiding d) where
2   get_env = Just $ BindEnv amb_is_not_err
```

4. Applications

In Sect. 2, we described issues that we have encountered that led to the development of this framework. We also highlighted some positive progress that we have been able to make using this framework. For starters, besides uncovering misinterpretations of language specifications in current compilers, we can now clarify complex features like ADL and

uses of operators in C++, as well as differences between C++, Haskell, and proposed features like multimethods for C++ and TDNR for Haskell. This section is intended to precisely cover such progress, with Sects. 4.1 and 4.2 showing applications of our scope combinations, Sect. 4.3 showing applications of the `Language` concept, and Sect. 4.4 highlighting applications to compilers covered outside of this paper.

4.1 Understanding Argument-Dependent Lookup

```
1  namespace ns1 {
2    struct X {};
3    void foo(X);
4  }
5  namespace ns2 {
6    void foo(int);
7  }
8  void bar(ns1::X x) {
9    using ns2::foo;
10   foo(x);
11 }
12 void baz(ns1::X x) {
13   void foo(int);
14   foo(x);
15 }
```

Figure 4: ADL in C++

ADL (as specified in [51] clause 3.4.2) makes additional declarations available for name binding by introducing declarations from scopes where the types of the arguments are defined. Fig. 4 shows an example of ADL, similar to the earlier example in Fig. 2 (Line 25). As small of a program as this example shows, we can already start to notice some inconsistencies in the ways that compilers interpret the specifications of ADL.

Upon experiments with different C++ compilers, namely Clang 4.1 [8], GCC 4.8 [13], the Intel compiler *13.0.1* [22], Microsoft Visual Studio [31], and the Comeau compiler [9], we find that when it comes to ADL, the compilers behave inconsistently, with the Intel compiler [22] differing from the other compilers. The example shows two functions, `bar` and `baz`, both calling a func-

tion `foo` on an argument of type `ns1::X`. As per the example in Fig. 2, ADL should be enabled in `bar`, leading to a successful call, and disabled in `baz`, leading to an error stating that the call on Line 14 does not match the declaration on Line 13. GCC *4.8*, Clang *4.1*, and Comeau do indeed issue such error while the Intel compiler does not.

```
1 void zet(ns1::X x) {
2   void foo(int);
3   foo(x);
4   {
5     using ns2::foo;
6     foo(x);
7   }
8 }
```

Figure 5: ADL in C++, extended

Fig. 5 illustrates further complications with ADL. Assuming the namespaces from Fig. 4, the first call to `foo` fails because ADL is blocked by the declaration of `foo` on the line above. The call in the nested scope succeeds however. According to the ADL rules, if a name was found through a using declaration, then ADL proceeds. The first declaration of `foo` is hidden by the using directive, so ADL succeeds in resolving the second call to `foo`. We can express this set of rules, more generally, using the following scope expression [3]:

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

when the name binding is triggered from a block scope, or

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

[3] $\lhd_{i=1}^{n}$ represents an iteration of $\lhd$ over values of i ranging from 1 to n (inclusive).

when it is triggered from a namespace scope, where [4]

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

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

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

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

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

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

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

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

This generalized expression covers most of the standard, excluding qualified name lookup. The functions $\mathbf{f_{ADL}}$ and $\mathbf{fn_{ADL}}$ represent a further expansion of the scoping rules, which states that a declaration matching a function call is progressively searched for through block scopes, and then, equally, through all the surrounding namespaces, their `using` declarations, and associated namespaces (**ADL**). The distinction between $\mathbf{f_{ADL}}$ and $\mathbf{fn_{ADL}}$ indicates that function declarations

in block scopes may hide associated namespaces but those in the surrounding namespaces cannot. Also, namespace scopes may enable ADL but block scopes with no `using` declaration cannot. Finally, ADL can occur at block scopes even if one of the scopes disables ADL.

The complexity of ADL is a prime example that shows how our framework can be used to capture complex designs, that normally require multiple paragraphs of specification in the standard, in a concise form that aids in analysis and understanding.

4.2 Understanding C++ Operators

```
1  struct X {}; struct Y {};
2
3  void operator+(X, X) { }
4  void operator+(X, Y) { }
5
6  void test(X x, Y y) {
7    void operator+(X, X);
8    x + x;
9    x + y;
10   operator+(x, x);
11   operator+(x, y);
12 }
```

Figure 6: Using C++ operators

The use of C++ operators is another interesting portion of our experiments. At first, it appears to be a rather simple feature. However, not only does it quickly gain complexity with the addition of ADL, but its specification appears to be widely misinterpreted by production quality compilers. Take, for instance, the program in Fig. 6. Our experiments unveil that most compilers handle it incorrectly. Clang 4.1 [8] and GCC 4.8 [13] both signal an error on Line 11, stating that `y` cannot be converted to `X` as required by the declaration on Line 7. The Intel compiler *13.0.1* [22] accepts the code. Microsoft Visual Studio

[31] compiles the code, but its visual editor indicates an error (this is due to the editor using a different frontend than the compiler). The Comeau compiler [9], the only one we tested that handles this example correctly, rejects both the operator expression on Line 7 and the operator function call on Line 11.

Why do mature production-grade compilers produce so different results for such a seemingly simple piece of code? The specification of binding uses of operators ([51], clause

[4] Symbols such as **S**, **N**, **H** and **M** are simply annotations for different kinds of scopes, based on the wording of the C++ standard, that do not change the process of name binding itself, except for how they are combined.

13.3.1.2) is spread out over several paragraphs and contains references to other parts of the C++ specification. The correct approach is to treat operator function calls such as the ones on Lines 10–11 as normal function calls with all the usual hiding rules. Operator expressions such as on Lines 8–9 should be converted into operator function calls, according to a table given in the specification. Then, the functions are to be looked up according to modified lookup rules that merge member, non-member, and built-in candidates together, which is not the usual approach for functions. We have not analyzed the implementations of the different compilers to understand how the specification has been misunderstood or even if the errors are a known issue, but this example shows that a framework like ours can be an invaluable tool for clarifying how operators are looked up. Based on the specification, we specify the general scoping rules for operator lookup (after the operator is converted into an appropriate function call) as

$$\left(\mathbf{H} \Leftarrow \mathbf{U_H} \Leftarrow \mathbf{O_B} \Leftarrow \mathbf{M}\right)$$

$$\lhd \left\{\!\!\left\{ \right.\right._{i=1}^{s} \mathbf{S_i} \lhd \left\{\!\!\left\{ \right.\right._{i=1}^{n} \mathbf{N_i}$$

without ADL, or

$$\begin{aligned}&\Big(\mathbf{H} \Leftarrow \mathbf{U_H} \Leftarrow \mathbf{O_B} \Leftarrow \mathbf{M} \Leftarrow \\ &\quad \left(\mathbf{U_H} \rhd \left(\left(\mathbf{H} \Leftarrow \widetilde{\mathbf{U}}_{\mathbf{H}} \Leftarrow \mathbf{O_B} \Leftarrow \mathbf{M}\right) \lhd \mathbf{ADL}\right)\right)\Big) \\ &\lhd \left\{\!\!\left\{ \right.\right._{i=1}^{s} \mathbf{f_{ADL}}\left(\mathbf{S_i}\right) \lhd \left\{\!\!\left\{ \right.\right._{i=1}^{n-1} \mathbf{fn_{ADL}}\left(\mathbf{N_i}\right) \lhd\end{aligned}$$

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

with ADL, where

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

The above expression states that the scope of the call, the members of the first operand, and the built-in operators are considered first, and, if nothing was found in one of these scopes, the surrounding scopes are considered next.

One can reduce the expression by noting that $\widetilde{\mathbf{X}}$ must be empty since uses of operators lookup operator functions, which are named with the reserved keyword `operator`. After reduction, we get

$$\left(\mathbf{H} \Leftrightarrow \mathbf{U_H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M} \Leftrightarrow\right.$$

$$\Big(\mathbf{U_H} \rhd \Big(\Big(\mathbf{H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M}\Big) \lhd \mathbf{ADL}\Big)\Big)$$
$$\lhd \langle\!\!\langle_{i=1}^{s} \mathbf{f_{ADL}}\,(\mathbf{S_i}) \lhd \langle\!\!\langle_{i=1}^{n-1} \mathbf{fn_{ADL}}\,(\mathbf{N_i}) \lhd (\mathbf{N_n} \Leftrightarrow \mathbf{U_{N_n}} \Leftrightarrow \mathbf{ADL})$$

and then

$$\Big(\mathbf{H} \Leftrightarrow \mathbf{U_H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M} \Leftrightarrow \Big(\mathbf{U_H} \rhd \Big(\Big(\mathbf{H} \Leftrightarrow \mathbf{O_B} \Leftrightarrow \mathbf{M}\Big) \lhd \mathbf{ADL}\Big)\Big)$$
$$\lhd \langle\!\!\langle_{i=1}^{s} \mathbf{f_{ADL}}\,(\mathbf{S_i}) \lhd \Big(\langle\!\!\langle_{i=1}^{n} (\mathbf{N_i} \Leftrightarrow \mathbf{U_{N_i}}) \Leftrightarrow \mathbf{ADL}\Big)$$

where

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

4.3 A Cross-Language Analysis

Aside from combining scopes using our combinators, one can carry on substantial discussions about the process of

name binding solely based on its properties of `match`, `select_best_viable` and `ambiguity`; that is, by simply instantiating the `Language` concept. In this section, we demonstrate this with a representative series of comparative analyses between selected features of C++ and Haskell. The essential thing to notice with these analyses is that we never mention language-specific details, unless they are directly related to the salient properties. Explicit instantiations, along with the implementation of our framework, is available online [6].

C++ Types vs. Functions, in Clang: In C++, as in most languages, ambiguity is an error by default. So, function calls and uses of types at least share that property. However, when it comes to matching references to declarations or reducing overload sets, at least judging from the implementation structure of Clang [8], they follow different paths.

For `match`, both kinds of uses start off checking that declarations and references share the same name. Then, there is a normalization process that removes all duplicates from the set of matching declarations, and even filters out unacceptable template declarations as necessary. The main point of difference comes from Clang's implementation of the normalization, in `LookupResult::resolveKind()`. When more than one distinct declarations are found, C++ considers it an ambiguity if the reference is not a function call; and an overload otherwise. So, for non functions, Clang instantly marks the lookup result as ambiguous. The effect of this is that, only for non-functions, Clang performs some name resolution during name lookup.

For `select_best_viable`, functions can be overloaded,

so this simply consists of overload resolution. Types, on the other hand cannot be overloaded. So, this starts off by checking for any kind of ambiguity in the lookup result. In Clang, name lookup, through `match`, has already performed this check and flagged the lookup result appropriately. So, `select_best_viable` simply checks for the flag. If the lookup result is either ambiguous or empty, `select_best_viable` is identity and thus immediately passes the result through to `assess`. If not, it performs a viability check on the single matching declaration, as necessary, and either passes the declaration onto `assess` if the viability check succeeds, or nothing otherwise.

```
1 struct X, Y, Z;
2 void foo(virtual X&, virtual Y&); // (1)
3 void foo(virtual Y&, virtual Y&); // (2) - **
      unique-base method **
4 void foo(virtual Y&, virtual Z&); // (3)
5 struct XY : X, Y {}
6 struct YZ : Y, Z {}
7 void foo(virtual XY&, virtual Y&); // (4) -
      overrider for (1) and (2)
8 void foo(virtual Y&, virtual YZ&); // (5) -
      overrider for (2) and (3)
9
10 XY xy; YZ yz;
11 foo(xy,yz);  // both (4) and (5) are equally
      viable matches, with unique base (2)
```

Figure 7: C++ multimethods example

The essential take from this analysis is that by talking about the process of name binding in terms of the salient properties in the `Language` concept, we have revealed subtleties in the implementation of uses of types that do not apply to uses of functions; and which may indicate just how composable components of their respective implementations are.

When Ambiguity Is Not an Error: It is unusual for ambiguity to not be an error. However, there exists special circumstances under which this kind of relaxed property is just what a language needs. Take, for instance, C++'s proposed extension with multimethods [35]. Fig. 7 shows an example that is taken straight from the proposal.

The proposal discusses semantic and engineering bene-

fits to relaxing the rules for ambiguity in such a way that the call in Line 11, which is normally considered ambiguous, would be considered valid. In essence, it allows ambiguous calls when all best viable candidates, called *overriders*, "have a unique base method through which the call can be dispatched". In such cases, it argues for using the unique base to seed the runtime dispatch table.

While the proposal covers greater details than illustrated here, we simply use this to illustrate the wide range of applicability of our framework, as well as to reinforce why ambiguity is a salient property of the language. For the example in Fig. 7, we find it reasonable to assume that `match` and `select_best_viable` remain as defined in C++ sans multimethods.

Type-Directed Name Resolution vs. Overloading: TDNR [50] is a proposed Haskell extension that "exploits the power of the dot" to provide an alternative way to disambiguate name uses. In essence, if a function call `f a` matches two separate declarations of different types, TDNR allows one to select a declaration based on the type of the call argument using a syntax like `a.f`. The similarity of this syntax to that of object-oriented languages has raised questions about how TDNR relates to the notion of overloading. Granted there has been several discussions over this within the Haskell community, our framework provides an alternative way to carry on the discussions. As a guide, we use it here to show how TDNR is not overloading based on the language's definition of name binding.

First and foremost, TDNR does not affect either of the `select_best_viable` or `ambiguity` salient properties of name resolution, whether type inference is part of the pro-

cess or not. Therefore, it cannot be overloading in the sense of *ad-hoc* polymorphism. It is not overloading based on type-class constraints because the call `a.f` is subject to the same scoping rules as `f a`, when expressed using our scope com-

binators (Sects. 3.1 and 3.2). The only difference between the calls `a.f` and `f a` is in the `match` property. For any function call, `match` typically just checks that the names of declarations match that of the call. However, for the dotted call `a.f`, it adds a normalization step that filters declarations that do not match the type of `a` out of consideration. The type of `a` is deduced before the binding of `a.f` is triggered, and is used by `match` for the said binding. So, there is no reason for any of the other components to be involved.

4.4 Compiler Integration

Our framework can facilitate automated reasoning in compiler design. Compilers can either

- implement the scoping rules of a given language entirely based on our scope combinators (Sects. 3.1 and 3.2), or
- simply extend their current mechanism for name binding with only weak hiding (Sect. 3.2).

Work is currently underway that explores and implements all these alternatives for reduced versions of practical languages like language-c, featherweight Java, and a mini-C++ language that we designed (cf. [6]). Particularly noteworthy,

an implementation of weak hiding is based on iteratively applying current mechanisms for name binding, noting that binding a scope expression $\mathbf{s_1} \Lleftarrow \mathbf{s_2}$ is equivalent to binding $\mathbf{s_1}$, followed by binding $\mathbf{s_2}$ if unsuccessful. That is,

$$\begin{aligned} &\textbf{\textit{bind}}\ (\mathit{ref}, (\mathbf{s_1} \Lleftarrow \mathbf{s_2})) \quad \equiv \\ &\qquad \textbf{\textit{bind}}\ (\mathit{ref}, \mathbf{s_1})\ \textbf{?}\ \textbf{\textit{bind}}\ (\mathit{ref}, \mathbf{s_1})\ \textbf{:}\ \textbf{\textit{bind}}\ (\mathit{ref}, \mathbf{s_2})\ . \end{aligned}$$

This implementation is called ***2**-stage name **bind**ing* ($Bind^{\times 2}$) for obvious reasons, has automated capabilities, and is currently implemented in ConceptClang [53].

5. Discussion and Related Work

We have introduced a framework to help carry on analytical discussions about name binding, and even its implementation, without bogging down to unnecessary details. This was necessary since a representative mini survey of well-known programming languages books shows that there is not a widely accepted formalism to reason about it. Scott [41] offers one of the most complete treatments of name binding. He introduces *bindings*, *scope rules*, and *overloading* where the results of *lookup* are followed by a static *analyzer* choosing the best declaration. Scott also frequently uses the term *reference* for name uses. Sebesta [42] provides a very similar discussion. Friedman et al. [12] discuss "scoping and binding variables," but do not include overloading. Abelson and Sussman [1] introduce names that *refer* to variables and the *environment* to keep the association between variables and their values. Pierce [34] only discusses name binding implicitly, as components of the specific systems discussed in the

book. Mitchell [32] also introduces the notion of scopes, but his description is mostly about implementation aspects. In summary, there is no standard way to specify name lookup – or name binding – independently of language details.

Applications of our framework are not limited to lexical scoping. In fact, for dynamic scoping, we can use our scope combinators to express the scoping rules for binding any given reference in a manner similar to the formulations presented in this paper (e.g. Eqs. 1-3, Sects. 4.1 and 4.2). The only difference from lexical scoping is that the scopes are collected and combined based on the calling stack rather than the lexical visibility of names when they are parsed.

Our framework provides a unified way to specify name binding, but can also benefit from formal extensions based on previous and related work [21, 23, 43, 44, 56, 57, 59, 60], as well as structural extensions based on ongoing work [6]. As structural extensions, we may consider alternative definitions of the scope combinators or of the `Language` concept, including a different compositional view of name binding altogether. As formal extensions, it helps to note that of all previous and related work that we have surveyed, only one has actually defined semantics for checking name uses in restricted scopes [58, 60]. This work builds up on the other works on formalizing concepts and characterizes the checking of name uses within restricted scopes as a name lookup problem. It then formalizes the said checking as consisting of name lookup within the constraints only. As such, in relation to our framework, we find it limited in three ways. First,

1. it is based on a specific design of concepts, the pre-Frankfurt design [5], for a specific language, C++. Second,

2. it restricts the checking of name uses within constraints to name lookup within the constraints only, thereby ignoring outer scopes. Third,
3. it does not support the *weak hiding* scoping rule, and perhaps only implicitly supports the *opening* scoping rule.

Moving forward, we may combine ideas from this work and [58, 60] and the formalization of C++ templates [44] to at least formalize *weak hiding* or $Bind^{\times 2}$. Alternatively, and for the other unconventional scoping rule of *opening*, we may consider extending $System\ F$, or even $System\ F^G$ [43] with all novel components of our framework.

Upon further investigation, we have noted that a large portion of related work comes from the area of *name analysis* or *name management* [7, 10, 11, 17, 24–26, 28, 29, 36–40, 48, 55]. However, this area tends to take fundamentally different approaches than necessary for our framework. In particular, it is centered around issues with engineering name binding (e.g. [29, 55]), while our framework is more interested in expressing name binding. A representative survey of related work follows.

Attribute grammars [28] naturally support the notion of symbol table construction through grammar attributes. Since introduction by Knuth, attribute grammars have been used in many variations. For example, Rewritable Reference Attribute Grammars (ReRAGs) [10] have been used for construction of symbol tables for context-sensitive languages and restructuring these symbol tables for refactoring of names. ReRAGs have been used to define sound renaming for Java [40]. The basic tenet of this work is the ability to provide provably correct inversion of name lookup matching declarations to name uses from the definition of

name lookup alone. Ekman and Hedin [11] demonstrate how ReRAGs can be used to structure name analysis in a declarative and modular way, structuring the implementation similarly to the specification. They deal with similar issues that our framework tackles, such as ambiguity and nested scopes with name hiding. Their work, however, does not provide a formal way to consider and specify lookup independently of implementations; they are more concerned with a modular implementation than with a tool to reason about designs for scoping rules. Kastens and Waite [26] tackle the problem of name analysis by encapsulating the basic operations necessary to define it in an abstract data type (ADT). Their ADT is a lookup table with scoping rules, providing operations to create scopes, insert and lookup names, and so on. The ADT can then be used in an attribute grammar where contextual analysis is specified. Our framework could also be used in a similar way, but we separate Kasten and Waite's ADT into

scopes, scope ordering relations, and an ambiguity design. Furthermore, our framework provides a complete solution for describing all features of name analysis, including overloading, while their framework is only concerned with name lookup and is supposed to be used with a separate overloading module. Kasten and Waite's ADT is used in the Eli compiler construction system [17], which provides a library of modules for interpreting the lookup table created in an attribute grammar (e.g., multiple inheritance module).

Reiss [38] provides a complete system for "symbol processing" in the ACORN automatic compiler production project [37] based on attribute grammar approach. Reiss provides a formal name binding model with some components similar to ours, such as LOOK_UP and RESOLVE functions. Symbol tables for particular languages are specified using an abstract description language. A symbol table can then be processed either through a low-level interface or through a domain-specific language in an attribute grammar framework. Reiss's system is similar to ours in that it recognizes a formal model for name lookup, which shares many common elements with our framework. In difference to our approach, Reiss's system is meant for implementing a language rather than abstracting from it. Because of that, even the abstract description of a symbol lookup module resembles implementation of a compiler. Our framework is better suited for design and analysis, yet it still provides a useful implementation framework.

Name management [24, 25] has been introduced as one of the important computer science issues for 1980's [39] and was defined to be the means for establishing names for objects, accessing the objects using their names, and controlling the availability and the meaning of names in different scopes. Name management shares some similar basic terminology with our approach, but its goals are quite different. In particular, name management is meant to alleviate the issues of interoperability and performance, and, as such, is more concerned with modeling of implementation of names, objects, and access, introducing notion of time of scope duration, for instance. Name management could be used as a general "backend" for our framework instead of a particu-

lar programming language, providing a layer of insulation between our abstractions and particular languages.

Power and Malloy [36] describe name lookup in C++ as a collection of UML [7] artifacts. Their description concentrates on name lookup explicitly, and their goal is to simplify implementation of future C++ tools by describing one of the hard implementation issues on the high level of a UML specification. While they isolate name lookup like we do, their primary concern is specification of implementations of name lookup for C++ rather than an abstract framework for making of design decisions.

Sulzmann [48] introduced a general framework for Hindley-Milner type systems with constraints. His work allows consideration of various language extensions as instantiations of the general framework, isolating the concern of a constraint system from the rest of the language. The framework is based on an idealized formal Hindley-Milner system parameterized by a constraint system. Our work is parameterized in the opposite direction: we provide a framework that is built on an abstraction of a language contained in an opaque interface. Our framework is applied by taking a language and providing appropriate instances in our framework, including a "compiler" component that generates scope descriptors for references. Then, the language uses our framework for name binding.

Konat et *al.* [29] pursue similar objectives to our work, i.e., describing name binding and scoping rules in a language-parametric fashion. However, their work quickly dives into implementation details that are not necessary for our purposes. In particular, in addition to abstracting references, declarations and scopes, they abstract namespaces and imports. In our framework, namespaces are simply scopes that do not need special consideration, except for clarifying scope expressions with respect to the wording of the standard document; and imports are expressed via the scope combinators – usually *merging* – based on the language. Moreover, using their framework requires defining binding rules in the Stratego rewriting language [52] or for the Spoofax language workbench [27], thereby specifying details in the kinds of scopes (e.g. namespaces, classes, etc...), declarations (e.g. classes, methods, variables, imports, unique, etc...), or references (e.g. uses of classes, types, variables, fields, etc...). In contrast, using our framework, one can talk about different ways to perform name binding, especially in restricted scopes, without defining such binding rules. There are two other points of differences. One point is that their framework is limited to lexical scoping whereas ours is not. The other point is that they abstract from the process of name binding, defining it as dependant on some language-specific "*name resolution strategy*"; while we segment the process further into two distinct components of *lookup* and *resolution*, which allows us to distinctly define the *weak hiding* scoping rule (independently of the language). Both frameworks may share similar goals, but the fundamental motivations and structures are different yet likely complementary. In fact, the level of detail that Konat et *al.*'s framework addresses may be necessary to instantiate our `Language` concept.

6. Conclusion

We have designed a framework for understanding and specifying name binding, independently of language-specific details. The framework abstracts the fundamental notions of *references*, *declarations* and *scopes*, and consists of two layers of abstractions: the first layer allows to express scoping rules using four combinators that we have defined, named *hiding*, *merging*, *opening* and *weak hiding*; and the second layer allows to express other properties of name binding not captured by the combinators that we abstracted in a concept named `Language`. Using this framework, one can carry on analysis and experimentations of name binding across different languages, features, and even kinds of references in a unified and structured fashion. For instance, we uncovered misinterpretations of the specifications for ADL and uses of operators in C++, concisely expressed what specifications actually say for various features like ADL (in C++) and TDNR (in Haskell), and performed structured cross-language analysis.

One particular scoping rule that we defined, *weak hiding*, is new in programming languages and useful to ensure the backward compatibility of C++ programs as C++ transitions into ConceptC++. Supporting *weak hiding* in compilers entails a new mechanism for name binding, $Bind^{\times 2}$, which simply consists of an iterative process over current mechanisms for name binding. $Bind^{\times 2}$ is currently implemented in ConceptClang, as *weak hiding* is considered essential for implementing concepts for C++.

As evidenced in this paper, in an ideal world, specifications could be formulated based on our framework, and

compilers could unambiguously implement name binding using such formulation.

Acknowledgments

Jeremiah Wilcock participated in earlier versions of this paper. We owe the idea of defining the scoping rules as combinators to discussions with Jeremiah and feedback from earlier submission reviews. We would also like to thank Jeremy Siek, Ryan Newton, Arun Chauhan, and Daniel Leivant for insightful comments that have led to this paper.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, Sept. 1996.

[2] D. Abrahams. Explicit namespaces. Technical Report N1691=04-0131, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, sep 2004.

[3] D. Abrahams. Controlling argument-dependent lookup. Technical Report N3490=12-0180, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, oct 2012.

[4] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[5] Becker P. (ed.). Working Draft, Standard for Programming Language C++. Technical Report N2914=09-0104, ISO/IEC

JTC1/SC22/WG21—The C++ Standards Committee, June 2009.

[6] BindIt. The name binding framework. https://gitlab.crest.iu.edu/lvoufo/bindit, Feb. 2013.

[7] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[8] Clang. A C language family frontend for LLVM. http://clang.llvm.org/, 2012.

[9] Comeau. Test Drive Comeau C++ Online!! http://www.comeaucomputing.com/tryitout/, Dec. 2012.

[10] T. Ekman and G. Hedin. Rewritable reference attributed grammars. *ECOOP 2004–Object-Oriented Programming*, pages 125–154, 2004.

[11] T. Ekman and G. Hedin. Modular Name Analysis for Java Using JastAdd. In Lämmel, Ralf and Saraiva, João and Visser, Joost, editor, *Generative and Transformational Techniques in Software Engineering*, number 4143 in LNCS, pages 422–436. Springer, Jan. 2006.

[12] D. Friedman, M. Wand, and C. Haynes. *Essentials of Programming Languages*. MIT Press, 2nd edition, 2001.

[13] GCC. The GNU Compiler Collection. http://gcc.gnu.org/, Dec. 2012.

[14] GNU Project Team. The gnu c++ library – 4.6 release – stl rotate() for ramdom access iterators. http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-api-4.6/a01048_source.html#l01571.

[15] GNU Project Team. The gnu c++ library (2011). http://gcc.gnu.org/onlinedocs/libstdc++/, Dec. 2012.

[16] P. Gottschling. Simplifying argument-dependent lookup rules. Technical Report N3595, ISO/IEC JTC 1, Information Technol-

ogy, Subcommittee SC 22, Programming Language C++, mar 2013.

[17] R. Gray, S. Levi, V. Heuring, A. Sloane, and W. Waite. Eli: A Complete, Flexible Compiler Construction System. *Commun. ACM*, 35(2):121–130, 1992.

[18] D. Gregor. Type-Soundness and optimization in the concepts proposal. Technical Report N2576=08-0086, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Mar. 2008.

[19] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 291–310. ACM Press, 2006.

[20] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in c++. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. . URL `http://doi.acm.org/10.1145/1167473.1167499`.

[21] M. Haveraaen. Institutions, property-aware programming and testing. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 21–30. ACM, 2007.

[22] Intel. Composer XE 2013. `http://software.intel.com/en-us/intel-composer-xe/`, Dec. 2012.

[23] J. Järvi, J. Willcock, and A. Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 1–19. ACM Press, 2005.

[24] A. Kaplan. *Name Management in Convergent Computing Systems: Models, Mechanisms, and Applications*. PhD thesis, University of

Massachusetts, May 1996.

[25] A. Kaplan and J. C. Wileden. Formalization and application of a unifying model for name management. In *Proc. 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 161–172. ACM Press, 1995.

[26] U. Kastens and W. M. Waite. An Abstract Data Type for Name Analysis. *Acta Informatica*, 28(6):539–558, June 1991.

[27] L. C. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.

[28] D. Knuth. Semantics of Context-Free Languages. *Theory Comput. Syst.*, 2(2):127–145, 1968.

[29] G. Konat, L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *Software Language Engineering*, pages 311–331. Springer, 2013.

[30] S. Marlow. Haskell 2010 Language Report. `http://www.haskell.org/onlinereport/haskell2010/`, 2010.

[31] Microsoft. Visual Studio. http://www.microsoft.com/visualstudio, Dec. 2012.

[32] J. C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 1st edition, Oct. 2002.

[33] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison-Wesley, 2nd edition, 2001.

[34] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[35] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Open Multi-Methods for C++. In *Proc. 6th Int. Conf. on Generative Programming and Component Engineering (GPCE)*, pages 123–134. ACM, 2007.

[36] J. Power and B. Malloy. Symbol Table Construction and Name Lookup in ISO C++. In *37th Int. Conf. on Technology of Object-*

Oriented Languages and Systems (TOOLS), pages 57–68. IEEE Comput. Soc., Nov. 2000.

[37] S. Reiss. Automatic Compiler Production: The Front End. *IEEE Trans. Softw. Eng.*, (6):609–627, 1987.

[38] S. P. Reiss. Generation of compiler symbol processing mechanisms from specifications. *ACM Trans. Program. Lang. Syst.*, 5 (2):127âĂŞ163, Apr. 1983.

[39] L. Rowe, P. Deutsch, S. Feldman, B. Lampson, B. Liskov, and T. Winograd. Programming Language Issues for the 1980's: SIGPLAN'83: Symposium on Programming Languages Issues in Software Systems. *SIGPLAN Not.*, 19(8):51–61, 1984.

[40] M. Schäfer, T. Ekman, and O. de Moor. Sound and Extensible Renaming for Java. *SIGPLAN Not.*, 43(10):277–294, 2008.

[41] M. L. Scott. *Programming Language Pragmatics, Second Edition*. Morgan Kaufmann, 2 edition, Nov. 2005.

[42] R. W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, 10th edition, 2012.

[43] J. G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, August 2005. AAI3183499.

[44] J. G. Siek and W. Taha. A semantic analysis of C++ templates. In *ECOOP 2006: European Conference on Object-Oriented Programming*, July 2006.

[45] G. L. Steele. *Common LISP: the language*. Digital press, 1990.

[46] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, May 1994. revised in October 1995 as tech. rep. HPL-95-11.

[47] B. Stroustrup and A. Sutton. A Concept Design for the STL. Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Jan. 2012.

[48] M. Sulzmann. *A General Framework for Hindley/Milner Type System with Constraints*. PhD thesis, Yale University, 2000.

[49] H. Sutter. A modest proposal: Fixing adl (revision 2). Technical Report N2103=06?0173, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, oct 2006.

[50] TDNR. Proposal: Typedirectednameresolution. `https://ghc.haskell.org/trac/haskell-prime/wiki/TypeDirectedNameResolution`, July 2009.

[51] S. D. Toit. Working Draft, Standard for Programming Language C++. Technical Report N3337, ISO/IEC JTC1/SC22/WG21—The C++ Standards Committee, Nov. 2012.

[52] E. Visser. Program transformation with stratego/xt. In *Domain-Specific Program Generation*, pages 216–238. Springer, 2004.

[53] L. Voufo. ConceptClang Project. `http://www.crest.iu.edu/projects/conceptcpp/`, May 2012.

[54] L. Voufo, M. Zalewski, and A. Lumsdaine. ConceptClang: an implementation of C++ concepts in Clang. In *Proc. 7th ACM SIGPLAN workshop on Generic programming*, pages 71–82. ACM Press, 2011.

[55] G. H. Wachsmuth, G. D. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. Language-parametric incremental name and type analysis. *SLE*, 2013.

[56] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages

60–76. ACM, 1989.

[57] J. Willcock, J. Järvi, A. Lumsdaine, and D. Musser. A Formalization of Concepts for Generic Programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit*. Adobe Systems, Apr. 2004.

[58] M. Zalewski. A semantic definition of separate type checking in C++ with concepts. Abstract syntax and complete semantic definition. Technical Report 2008:12, Department of Computer Science and Engineering, Chalmers University, 2008.

[59] M. Zalewski and S. Schupp. C++ concepts as institutions: A specification view on concepts. In *Proc. Symposium on Library-Centric Software Design*, pages 76–87. ACM, 2007.

[60] M. Zalewski and S. Schupp. A semantic definition of separate type checking in C++ with concepts. *Journal of Object Technology*, 8(5):105–132, 2009. Extended version available in Zalewski's PhD thesis.