Declarative Name Binding and Scope Rules

Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser

Delft University of Technology, The Netherlands
g.d.p.konat@student.tudelft.nl,
{l.c.l.kats,g.h.wachsmuth,e.visser}@tudelft.nl

Abstract. In textual software languages, names are used to reference elements like variables, methods, classes, etc. Name resolution analyses these names in order to establish references between definition and use sites of elements. In this paper, we identify recurring patterns for name bindings in programming languages and introduce a declarative meta-language for the specification of name bindings in terms of namespaces, definition sites, use sites, and scopes. Based on such declarative name binding specifications, we provide a language-parametric algorithm for static name resolution during compile-time. We discuss the integration of the algorithm into the Spoofax Language Workbench and show how its results can be employed in semantic editor services like reference resolution, constraint checking, and content completion.

1 Introduction

Software language engineering is concerned with *linguistic abstraction*, the formalization of our understanding of domains of computation in higher-level software languages. Such languages allow direct expression in terms of the domain, instead of requiring encoding in a less specific language. They raise the level of abstraction and reduce accidental complexity. One of the key goals in the field of language engineering is to apply these techniques to the discipline itself: high-level languages to specify all aspects of software languages. Declarative languages are of particular interest since they enable language engineers to focus on the *What?* instead of the *How?*. Syntax definitions are a prominent example. With declarative formalisms such as EBNF, we can specify the syntactic concepts of a language without specifying how they can be recognized programmatically. This declarativity is crucial for language engineering. Losing it hampers evolution, maintainability, and compositionality of syntax definitions [15].

Despite the success of declarative syntax formalisms, we tend to programmatic specifications for other language aspects. Instead of specifying languages, we build programmatic language processors, following implementation patterns in rather general specification languages. These languages might still be considered domain-specific, when they provide special means for programmatic language processors. They also might be considered declarative, when they abstract over computation order. However, they enable us only to implement language

K. Czarnecki and G. Hedin (Eds.): SLE 2012, LNCS 7745, pp. 311–331, 2013.

processors faster, but not to specify language aspects. They lack domain concepts for these aspects and focus on the *How?*. That is a problem since (1) it entails overhead in encoding concepts in a programming language and (2) the encoding obscures the intention; understanding the definition requires decoding.

Our goal is to extend the set of really declarative, domain-specific languages for language specifications. In this paper, we are specifically concerned with *name binding and scope rules*. Name binding is concerned with the relation between definitions and references of identifiers in textual software languages, including scope rules that govern these relations. In language processors, it is crucial to make information about definitions available at the references. Therefore, traditional language processing approaches provide programmatic abstractions for name binding. These abstractions are centered around tree traversal and information propagation from definitions to references. Typically, they are not specifically addressing name binding, but can also be used for other language processing tasks such as compilation and interpretation.

Name binding plays a role in multiple language engineering processes, including editor services such as reference resolution, code completion, refactorings, type checking, and compilation. The different processes need different information about definitions. For example, name resolution tries to find one definition, while code completion needs to determine all possible references in a certain place. The different requirements lead either to multiple re-implementations of name binding rules for each of these purposes, or to non-trivial, manual weaving into a single implementation supporting all purposes. This results in code duplication with as result errors, inconsistencies, and increased maintenance effort.

The traditional paradigm influences not only language processing, but also language specification. For example, the OCL language standard [19] specifies name binding in terms of nested environments, which are maintained in a tree traversal. The C# language specification [1] defines name resolution as a sequence of imperative lookup operations. In this paper, we abstract from the programmatic mechanics of name resolution. Instead, we aim to declare the roles of language constructs in name binding and leave the resolution mechanics to a generator and run-time engine. We introduce the *Name Binding Language* (NBL), a language with linguistic abstractions for declarative definition of name binding and scope rules. NBL supports the declaration of definition and use sites of names, properties of these names associated with language constructs, namespaces for separating categories of names, scopes in which definitions are visible, and imports between scopes.

NBL is integrated in the Spoofax Language Workbench [14], but can be reused in other language processing environments. From definitions in the name binding language, a compiler generates a language-specific name resolution strategy in the Stratego rewriting language [25] by parametrizing an underlying generic, language independent strategy. Name resolution results in a persistent symbol table for use by semantic editor services such as reference resolution, consistency checking of definitions, type checking, refactoring, and code generation. The implementation supports multiple file analysis by default.

We proceed as follows. In Sect. 2 and 3 we introduce NBL by example, using a subset of the C# language. In Sect. 4 we discuss the derivation of editor services from a name binding specification. In Sect. 5 we give a high-level description of the generic name resolution algorithm underlying NBL. In Sect. 6 we discuss the integration of NBL into the Spoofax Language Workbench. Sect. 7 and 8 are for evaluation and related work.

2 Declarative Name Binding and Scope Rules

In this section we introduce the Spoofax Naming Binding Language illustrated with examples drawn from the specification of name binding for a subset of C# [1]. Fig. 1 defines the syntax of the subset in SDF [24]. The subset is by no

```
Using* NsMem*                                  → CompilationUnit {"Unit"}

"using" NsOrTypeName ";"                       → Using           {"Using"}
"using" ID "=" NsOrTypeName                    → Using           {"Alias"}
ID                                             → NsOrTypeName    {"NsOrType"}
NsOrTypeName "." ID                            → NsOrTypeName    {"NsOrType"}

"namespace" ID "{" Using* NsMem* "}"           → NsMem           {"Namespace"}
Partial "class" ID Base "{" ClassMem* "}"      → NsMem           {"Class"}

                                               → Partial         {"NonPartial"}
"partial"                                      → Partial         {"Partial"}
                                               → Base            {"NoBase"}
":" ID                                         → Base            {"Base"}

Type ID ";"                                    → ClassMem        {"Field"}
RetType ID "(" {Param ","}* ")" Block ";"      → ClassMem        {"Method"}

ID                                             → Type            {"ClassType"}
"int"                                          → Type            {"IntType"}
"bool"                                         → Type            {"BoolType"}
Type                                           → RetType
"void"                                         → RetType         {"Void"}
Type ID                                        → Param           {"Param"}

"{" Stmt* "}"                                  → Block           {"Block"}
Decl                                           → Stmt
EmbStmt                                        → Stmt
"return" Exp ";"                               → Stmt            {"Return"}
Type ID ";"                                    → Decl            {"Var"}
Type ID "=" Exp ";"                            → Decl            {"Var"}
Block                                          → EmbStmt
StmtExp ";"                                    → EmbStmt
"foreach" "(" Type ID "in" Exp ")" EmbStmt     → EmbStmt         {"Foreach"}

INT                                            → Exp             {"IntLit"}
"true"                                         → Exp             {"True"}
"false"                                        → Exp             {"False"}
ID                                             → Exp             {"VarRef"}
StmtExp                                        → Exp
Exp "." ID                                     → StmtExp         {"FieldAccess"}
Exp "." ID "(" {Exp ","}* ")"                  → StmtExp         {"Call"}
ID "(" {Exp ","}* ")"                          → StmtExp         {"Call"}
```

Fig. 1. Syntax definition in SDF for a subset of C#. The names in the annotations are abstract syntax tree constructors.

means complete; it has been selected to model representative features of name binding rules in programming and domain-specific languages. In the following subsections we discuss the following fundamental concepts of name binding: *definition and use sites*, *namespaces*, *scopes*, and *imports*. For each concept we give a general definition, illustrate it with an example in C#, and then we show how the concept can be modeled in NBL.

2.1 Definitions and References

The essence of name binding is establishing relations between a *definition* that *binds* a name and a *reference* that *uses* that name. Name binding is typically defined programmatically through a *name resolution algorithm* that connects references to definitions. A *definition site* is the location of a definition in a program. In many cases, definition sites are required to be *unique*, that is, there should be exactly one definition site for each name. However, there are cases where definition sites are allowed to be *non-unique*.

Example. Figure 2 contains class definitions in C#. Each class definition binds the name of a class. Thus, we have definition sites for `C1`, `C2`, and `C3`. Base class specifications are references to these definition sites. In the example, we have references to `C1` as the base class of `C2` and `C2` as the base class of `C3`. (Thus, `C2` is a sub-class of, or inherits from `C1`.) There is no reference to `C3`. The definition sites for `C1` and `C2` are unique. By contrast, there are two definition sites for `C3`, defining parts of the same class `C3`. Thus, these definition sites are non-unique. This is correct in C#, since regular class definitions are required to be unique, while partial class definitions are allowed to be non-unique.

```
class C1 {}
class C2:C1 {}
partial class C3:C2 {}
partial class C3 {}
```

Fig. 2. Class declarations in C#

Abstract Syntax Terms. In Spoofax abstract syntax trees (ASTs) are represented using first-order terms. Terms consist of strings (`"x"`), lists of terms (`["x","y"]`), and constructor applications (`ClassType("C1")`) for labelled tree nodes with a fixed number of children. Annotations in grammar productions (Fig. 1) define the constructors to be used in AST construction. For example, `Class(Partial(), "C3", Base("C2"), [])` is the representation of the first partial class in Figure 2. A term *pattern* is a term that may contain variables (`x`) and wildcards (`_`).

Model. A specification in the name binding language consists of a collection of rules of the form `pattern : clause*`, where `pattern` is a term pattern and `clause*` is a list of name binding declarations about the language construct that matches with `pattern`. Figure 3 shows a declaration of the definitions and references for class names in C#. The first two rules declare class definition sites for class names. Their patterns distinguish regular (non-partial) and partial class declarations. While non-partial class declarations are unique definition sites, partial class declarations are non-unique definition sites. The third rule declares that the term pattern `Base(c)` is a reference to a class with name `c`.

```
rules
  Class(NonPartial(), c, _, _): defines unique class c
  Class(Partial(), c, _, _)   : defines non-unique class c
  Base(c)                     : refers to class c
  ClassType(c)                : refers to class c
```

Fig. 3. Declaration of definitions and references for class names in C#

Thus, the `": C1"` in Figure 2 is a reference to class `C1`. Similarly, the second rule declares a class type as a reference to a class.

2.2 Namespaces

Definitions and references declare relations between named program elements and their uses. Languages typically distinguish several *namespaces*, i.e. different kinds of names, such that an occurrence of a name in one namespace is not related to an occurrence of that same name in another.

Example. Figure 4 shows several definitions for the same name `x`, but of different kinds, namely a class, a field, a method, and a variable. Each of these kinds has its own namespace in C#, and each of these namespaces has its own name `x`. This enables us to distinguish the definition sites of class `x`, field `x`, method `x`, and variable `x`, which are all unique.

```
class x {
  int x;
  void x() {
    int x; x = x + 1;
  }
}
```

Fig. 4. Homonym class, field, method, and variable declarations in C#

Model. We declared definitions and references for the namespace `class` already in the previous example. Figure 5 extends that declaration covering also the namespaces `field`, `method`, and `variable`. Note that it is required to declare namespaces to ensure the consistency of name binding rules. Definition sites are bound to a single namespace (**defines** *class* `c`), but use sites are not. For example, a variable in an expression might either refer to a variable, or to a field, which is modeled in the last rule. In our example, this means that variable declarations hide field declarations, because variables are resolved to variables, if possible. Thus, both `x` in the assignment in Figure 4 refer to the variable `x`.

2.3 Scopes

Scopes restrict the visibility of definition sites. A *named scope* is the definition site for a name which scopes other definition sites. By contrast, an *anonymous*

```
namespaces class field method variable
rules
  Field(_, f)         : defines unique field f
  Method(_, m, _, _): defines unique method m
  Call(m, _)          : refers to method m

  Var(_, v): defines unique variable v
  VarRef(x): refers to variable x otherwise to field x
```

Fig. 5. Declaration of name bindings for different namespaces in C#

```
1  class C {
2      void m() { int x; }
3  }
4
5  class D {
6    void m() {
7      int x;
8      int y;
9      { int x; x = y + 1; }
10     x = y + 1;
11   }
12 }
```

Fig. 6. Scoped homonym method and variable declarations in C#

```
rules
  Class(NonPartial(), c, _, _):
    defines unique class c
    scopes field, method

  Class(Partial(), c, _, _):
    defines non-unique class c
    scopes field, method

  Method(_, m, _, _):
    defines unique method m
    scopes variable

  Block(_): scopes variable
```

Fig. 7. Declaration of scopes for different namespaces in C#

scope does not define a name. Scopes can be nested and name resolution typically looks for definition sites from inner to outer scopes.

Example. Figure 6 includes two definition sites for a method `m`. These definition sites are not distinguishable by their namespace `method` and their name `m`, but, they are distinguishable by the scope they are in. The first definition site resides in class `C`, the second one in class `D`. In C#, class declarations scope method declarations. They introduce named scopes, because class declarations are definition sites for class names. The listing also contains three definition sites for a variable `x`. Again, these are distinguishable by their scope. In C#, method declarations and blocks scope variable declarations. Method declarations are named scopes, blocks are anonymous scopes. The first definition site resides in method `m` in class `C`, the second one in method `m` in class `D`, and the last one in a nameless block inside method `m` in class `D`. In the assignment inside the block (line 9), `x` refers to the variable declaration in the same block, while the `x` in the outer assignment (line 10) refers to the variable declaration outside the block. In both assignments, `y` refers to the variable declaration in the outer scope, because the block does not contain a definition site for `y`.

Model. The **scopes** `ns` clause in NBL declares a construct to be a scope for namespace `ns`. Figure 7 declares scopes for fields, methods, and variables. Named scopes are declared at definition sites. Anonymous scopes are declared similarly, but lack a `defines` clause.

Namespaces as Language Concepts. C# has a notion of 'namespaces'. It is important to distinguish these *namespaces as a language concept* from *namespaces as a naming concept*, which group names of different kinds of declarations. Specifically, in C#, namespace declarations are top-level scopes for class declarations. Namespace declarations can be nested. Figure 8 declares a top-level namespace `N`, scoping a class declaration `N` and an inner namespace declaration `N`.

```
namespace N {
  class N {}
  namespace N { class N {} }
}
```

Fig. 8. Nested namespace declarations in C#

```
namespaces namespace
rules
  Namespace(n, _):
    defines namespace n
    scopes namespace, class
```

Fig. 9. Declaration of name bindings for nested namespace declarations in C#

The inner namespace declaration scopes another class declaration `N`. The definition sites of the namespace name `N` and the class name `N` are distinguishable, because they belong to different namespaces (as a naming concept). The two definition sites of namespace name `N` are distinguishable by scope. The outer namespace declaration scopes the inner one. Also, the definition sites of the class name `N` are distinguishable by scope. The first one is scoped by the outer namespace declaration, while the second one is scoped by both namespace declarations.

Model. The names of C# namespace declarations are distinguishable from names of classes, fields, etc. As declared in Figure 9, their names belong to the `namespace` namespace. The name binding rules for definition sites of names of this namespace models the scoping nature of C# namespace declarations.

Imports. An import introduces into the current scope definitions from another scope, either under the same name or under a new name. An import that imports all definitions can be transitive.

Example. Figure 10 shows different kinds of imports in C#. First, a `using` directive imports type declarations from namespace `N`. Second, another `using` directive imports class `C` from namespace `M` into namespace `O` under a new name `D`. Finally, classes `E` and `F` import fields and methods from their base classes. These imports are transitive, that is, `F` imports fields and methods from `E` and `D`.

Model. Figure 11 shows name binding rules for import mechanisms in C#. The first rule handles `using` declarations, which import all classes from the namespace to which the qualified name `qname` resolves to. The second rule models aliases, which either import a namespace or a class under a new name, depending on the resolution of `qname`. The last rule models inheritance, where fields and methods are imported transitively from the base classes.

2.4 Types

So far, we discussed names, namespaces, and scopes to distinguish definition sites for the same name. Types also play a role in name resolution and can be used to distinguish definition sites for a name or to find corresponding definition sites for a use site.

```
using N;

namespace M {
  class C { int f; }
}

namespace O {
  using D = M.C;
  class E:D {
    void m() {}
  }
  class F:E { }
}
```

Fig. 10. Various forms of imports in C#

```
rules
  Using(qname):
    imports class from namespace ns
    where qname refers to namespace ns

  Alias(alias, qname):
    imports namespace ns as alias
    where qname refers to namespace ns
    otherwise imports class c as alias
    where qname refers to class c

  Base(c):
    imports field (transitive),
            method (transitive)
       from class c
```

Fig. 11. Declaration of import mechanisms in C#

Example. Figure 12 shows a number of overloaded method declarations. These share the same name `m`, namespace `method`, and scope class `C`. But we can distinguish them by the types of their parameters. Furthermore, all method calls inside method `x` can be uniquely resolved to one of these methods by taking the argument types of the calls into account.

```
class C {
  void m() {}
  void m(int x) {}
  void m(bool x) {}
  void m(int x, int y) {}
  void m(bool x, bool y) {}

  void x() {
    m();
    m(42);
    m(true);
    m(21, 21);
    m(true, false);
  }
}
```

Fig. 12. Overloaded method declarations in C#

Model. Figure 13 includes type information into name binding rules for fields, methods, and variables. Definition sites might have types. In the simplest case, the type is part of the declaration. In the example, this holds for parameters. For method calls, the type of the definition site for a method name depends on the types of the parameters. A type system is needed to connect the type of a single parameter, as declared in the rule for parameters, and the type of a list of parameters, as required in the rule for methods. We will discuss the influence of a type system and the interaction between name and type analysis later. For now, we assume that the type of a list of parameters is a list of types of these parameters.

Type information is also needed to resolve method calls to possibly overloaded methods. The `refers` clause for method calls therefore requires the corresponding definition site to match the type of the arguments. Again, we omit the details how this type can be determined. We also do not consider subtyping here. Method calls and corresponding method declarations need to have the same argument and parameter types.

3 Name Binding Patterns

We now identify typical name binding patterns. These patterns are formed by scopes, definition sites and their visibility, and use sites referencing these definition sites. We explain each pattern first and give an example in C# next. Afterwards, we show how the example can be modelled with declarative name binding rules.

```
rules
  Method(t, m, p*, _):
    defines unique method m of type (t*, t)
    where p* has type t*

  Call(m, a*):
    refers to method m of type (t*, _)
    where a* has type t*

  Param(t, p): defines unique variable p of type t
```

Fig. 13. Types in name binding rules for overloaded methods in C#

Unscoped Definition Sites. In the simplest case, definition sites are not scoped and globally visible.

Example. In C#, namespace and class declarations (as well as any other type declaration) can be unscoped. They are globally visible across file boundaries. For example, the classes `C1`, `C2`, and `C3` in Figure 2 are globally visible. In Figure 4, only the outer namespace `N` is globally visible.

In contrast to C#, C++ has file scopes and all top-level declarations are only visible in a file. To share global declarations, each file has to repeat the declaration and mark it as `extern`. This is typically achieved by importing a shared header file.

Model. We consider any definition site that is not scoped by another definition site or by an anonymous scope to be in global scope. These definition sites are visible over file boundaries. File scope can be modelled with a scoping rule in two different ways. Both are illustrated in Figure 14. The first rule declares the top-level node of abstract syntax trees as a scope for all namespaces which can have top-level declarations. This scope will be anonymous, because the top-level node cannot be a definition site (otherwise this definition site would be globally visible). The second rule declares a tuple consisting of file name and the abstract syntax tree as a scope. This tuple will be considered a definition site for the file name. Thus, the scope will be named after the file.

```
rules
  CompilationUnit(_, _):
    scopes namespace, class

  (f, CompilationUnit(_, _)):
    defines file f
    scopes namespace, class
```

Fig. 14. Different ways to model file scope for top-level syntax tree nodes

Definition Sites Inside Their Scopes. Typically, definition sites reside inside the scopes where they are visible. Such definition sites can either be visible only after their declaration, or everywhere in their surrounding scope.

Example. In C#, namespace members such as nested namespace declarations and class declarations are visible in their surrounding scope. The same holds for class members. In contrast, variable declarations inside a method scope become visible only after their declaration.

Model. Scoped definition sites are by default visible in the complete scope. Optionally, this can be stated explicitly in `defines` clauses. Figure 15 illustrates this for namespace declarations. The second rule in this listing shows how to model definition sites which become visible only after their declaration.

Definition Sites Outside Their Scopes Some declarations include not only the definition site for a name, but also the scope for this definition site. In such declarations, the definition site resides outside its scope.

```
rules
  Namespace(n, _):
    defines non-unique namespace n in surrounding scope

  Var(t, c):
    defines unique variable of type t in subsequent scope
```

Fig. 15. Declaration of the visibility of definition sites inside scopes

Example. Let expressions are a classical example for definition sites outside their scopes. In C#, `foreach` statements declare iterator variables, which are visible in embedded statement. Figure 16 shows a method with a parameter `x`, followed by a `foreach` statement with an iterator variable of the same name. This is considered incorrect in C#, because definition sites for variable names in inner scopes collide with definition sites of the same name in outer scopes. However, the use sites can still be resolved based on the scopes of the definition sites. The use site for `x` inside the loop refers to the iterator variable, while the `x` in the collection expression refers to the parameter.

```
class C {
  void m(int[] x) {
    foreach (int x in x)
      System.Console.WriteLine(x);
  }
}
```

Fig. 16. `foreach` loop with scoped iterator variable `x` in C#

Model. Figure 17 shows the name binding rule for `foreach` loops, stating the scope of the variable explicitly. Note that definition sites which become visible after their declaration are a special case of this pattern. Figure 18 illustrates how this can be modelled in the same way as the `foreach` loop. The first rule assumes a nested representation of statement sequences, while the second rule assumes a list of statements.

Contextual Use Sites. Definition sites can be referenced by use sites outside of their scopes. These use sites appear in a context which determines the scope into which they refer. This context can either be a direct reference to this scope, or has a type which determines the scope.

```
using N.N.N;

namespace N' {
  class C {
    C f;
    void m(C p) { }
  }
  class D {
    void m(C p) {
      p.m(p.f);
    }
  }
}
```

Fig. 19. Contextual use sites in C#

Example. In C#, namespace members can be imported into other namespaces. Figure 8 shows a class `N` in a nested namespace. In Figure 19, this class is imported. The `using` directive refers to the class with a qualified name. The first part of this name refers to the outer namespace `N`. It is the context of the second part, which refers to the inner namespace `N`. The second part is then the context for the last part of the qualified name, which refers to the class `N` inside the inner namespace.

```
rules
  Foreach(t, v, exp, body):
    defines unique variable v of type t in body
```

Fig. 17. Declaration of definition sites outside of their scopes

```
rules
  Seq(Var(t, v), stmts):
    defines unique variable v of type t in stmts

  [Var(t, v) | stmts]:
    defines unique variable v of type t in stmts
```

Fig. 18. Alternative declaration of definition sites becoming visible after their declaration

```
rules
  NsOrType(n1, n2):
    refers to namespace n2 in ns
    otherwise to class n2 in ns
    where n1 refers to namespace ns

  FieldAccess(e, f):
    refers to field f in c
    where e has type ClassType(c)

  MethodCall(e, m, p*):
    refers to method m of type (t*, _) in c
    where e has type ClassType(c)
    where p* has type t*
```

Fig. 20. Declaration of contextual use sites

Figure 19 also illustrates use sites in a type-based context. In method `m` in class `D`, a field `f` is accessed. The corresponding definition site is outside the scope of the method in class `C`. But this scope is given by the type of `p`, which is the context for the field access. Similarly, the method call is resolved to method `m` in class `C` because of the type of `p`.

Model. Figure 20 illustrates how to model contextual use sites. The scope of the declaration site corresponding to a use site can be modelled in `refers` clauses. This scope needs to be determined from the context of the use site. The first rule resolves the context of a qualified name part to a namespace `ns` and declares the use site to refer either to a namespace or to a class in `ns`. The remaining rules declare use sites for field access and method calls. They determine the type of the context, which needs to be a class type. A field access refers to a field in that class. Similarly, a method call refers to a method with the right parameter types in that class.

4 Editor Services

Modern IDEs provide a wide range of editor services where name resolution plays a large role. Traditionally, each of these services would be handcrafted for each language supported by the IDE, requiring substantial effort. However, by accurately modeling the relations between names in NBL, it is possible to generate a name resolution algorithm and editor services that are based on that algorithm.

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      posterName = "name";
7      string posterName;
8      posterName = user.nam;
9      string posterName = user.name;
10     return posterName;
11   }
12 }
```

Fig. 21. Error checking

Reference Resolving. Name resolution is exposed directly in the IDE in the form of reference resolving: press and hold Control and hover the mouse cursor over an identifier to reveal a blue hyperlink that leads to its definition side. This behavior is illustrated in Fig. 22.

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      string posterName;
7      posterName = user.name;
8      return posterName;
9    }
10 }
```

```
1  class User {
2    string name;
3  }
4  class Blog {
5    string post(User user, string message) {
6      string posterName;
7      posterName = user.name;
8      return posterName;
9    }
10 }
```

Fig. 22. Reference resolution of `name` field reference to `name` field definition

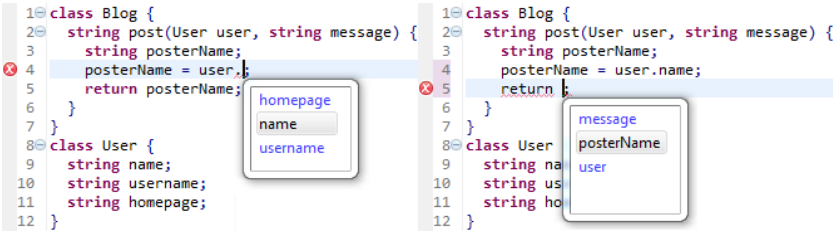

Fig. 23. Code completion for fields and local variables

Constraint Checking. Modern IDEs statically check programs against a wide range of constraints. Constraint checking is done on the fly while typing and directly displayed in the editor via error markers on the text and in the outline view. Error checking constraints are generated from the NBL for common name binding errors such as unresolved references, duplicate definitions, use before definition and unused definitions. Fig. 21 shows an editor with error markers. The `message` parameter in the `post` method has a warning marker indicating that it is not used in the method body. On the line that follows it, the `posterName` variable is assigned but has not yet been declared, violating the visibility rules of Figure 15. Other errors in the method include a subsequent duplicate definition of `posterName`, which violates the uniqueness constraint of the `variable` namespace of Figure 5, and referencing a non-existent property `nam`.

Code Completion. With code completion, partial (or empty) identifiers can be completed to full identifiers that are valid at the context where code completion is executed. Figure 23 shows an example of code completion. In the left program code completion is triggered on a field access expression on the `user` object. The `user` object is of type `User`, so all fields of `User` are shown as candidates. On the right, completion is triggered on a variable reference, so all variables in the current scope are shown.

5 Implementation

To implement name resolution based on NBL, we employ a name resolution algorithm that relies on a symbol table data structure to persist name bindings and lazy evaluation to resolve all references. In this section we give an overview of the data structure, the name resolution algorithm, and their implementation.

Persistence of Name Bindings. To persist name bindings, each definition and reference is assigned a qualified name in the form of a URI. The URI identifies the occurrence across a project. Use sites share the URIs of their corresponding definition sites.

A URI consists of the namespace, the path, and the name of a definition site. As an example, the URI `method://N/C/m` is assigned to a method `m` in a class `C` in a namespace `N`. Here, the segments represent the names of the

scopes. Anonymous scopes are represented by a special path segment `anon(u)`, where `u` is a unique string to distinguish different anonymous scopes. For use in analyses and transformations, URIs can be represented in the form of ATerms, e.g. `[method(),"N","C","m"]` is URI for the method `m`.

All name bindings are persisted in an in-memory data structure called the semantic index. It consists of a symbol table that lists all URIs that exist in a project, and can be efficiently implemented as a hash table. It maps each URI to the file and offset of their occurrences in the project. It can also store additional information, such as the type of a definition.

Resolving Names. Our algorithm is divided into three phases. First, in the annotation phase, all definition and use sites are assigned a preliminary URI, and definition sites are stored in the index. Second, definition sites are analyzed, and their types are stored in the index. And third, any unresolved references are resolved and stored in the index.

Annotation Phase. In the first phase, the AST of the input file is traversed in top-down order. The logical nesting hierarchy of programs follows from the AST, and is used to assign URIs to definition sites. For example, as the traversal enters the outer namespace scope `n`, any definitions inside it are assigned a URI that starts with '`n.`'. As a result of the annotation phase, all definition and use sites are annotated with a URI. In the case of definition sites, this is the definitive URI that identifies the definition across the project. For references, a temporary URI is assigned that indicates its context, but the actual definition it points to has to be resolved in a following phase. For reference by the following phases, all definitions are also stored in the index.

Definition Site Analysis Phase. The second phase analyzes each definition site in another top-down traversal. It determines any local information about the definition, such as its type, and stores it in the index so it can be referenced elsewhere. Types and other information that cannot be determined locally are determined and stored in the index in the last phase.

Use Site Analysis Phase. When the last phase commences, all local information about definitions has been stored in the index, and non-local information about definitions and uses in other files is available. What remains is to resolve references and to determine types that depend on non-local information (in particular, inferred types). While providing a full description of the use site analysis phase and the implementation of all name binding constructs is outside the scope of this paper, the below steps sketch how each reference is resolved. See the NBL website [1] for links to the algorithm's source files.

1. Determine the temporary URI `ns://path/n` which was annotated in the first analysis phase.
2. If an import exists in scope, expand the current URI for that import.
3. If the reference corresponds to a name-binding rule that depends on non-local information such as types, retrieve that information.

[1] `http://strategoxt.org/Spoofax/NBL`

4. Look for a definition in the index with namespace `ns`, path `path`, and name `n`. If it does not exist, try again with a prefix of `path` that is one segment shorter. If the no definition is found this way, store an error for the reference.
5. If the definition is an alias, resolve it.

An important part to highlight in the algorithm is the interaction between name and type analysis that happens for example with the `FieldAccess` expression of Figure 20. For name binding rules that depend on types or other non-local information, it is possible that determining the type recursively triggers name resolution. For this reason, we apply lazy evaluation, ensuring that any reference can be resolved lazily as requested in this phase. By traversing through the entire tree, we ensure that all use sites are eventually resolved and persisted to the index.

6 Integration into Spoofax

The NBL, together with the index, is integrated into the Spoofax Language Workbench. Stratego rules are generated by the NBL that use the index API to interface with Spoofax. In this section we will show the index API and how the API is used to integrate the editor services seen in Section 4.

Index API. Once all analysis phases have been completed, the index is filled with a summary of every file. To use the summaries we provide the index API with a number of lookups and queries. Lookups transform annotated identifiers into definitions. Queries transform definitions (retrieved using a lookup) into other data. The API is used for integrating editor services, but is also exposed to Spoofax language developers for specifying additional editor services or other transformations.

`index-lookup-one` performs a lookup that looks for a definition of given identifier in its owning scope. The `index-lookup` lookup performs a lookup that tries to look for a definition using `index-lookup-one`. If it cannot be found, the lookup is restarted on the outer scope until the root scope is reached. If no definition is found at the root scope, the lookup fails. There is also an `index-lookup-all` variant that returns all found definitions instead of stopping at the first found definition. Finally, `index-lookup-all-levels` is a special version of `index-lookup-all` that supports partial identifiers.

```
editor-complete:
  ast → identifiers
  where
    node@COMPLETION(name) := <collect-one(?COMPLETION(_))> ast ;
    proposals             := <index-lookup-all-levels(|name)> node ;
    identifiers           := <map(index-uri-name)> proposals
```

Fig. 25. Code completion

To get data from the index, `index-get-data` is used. Given a definition and a data kind, it will return all data values of that kind that is attached to the definition. Uses are retrieved in the same way using `index-get-uses-all`.

Reference Resolution. Resolving a reference to its definition is very straightforward when using `index−lookup`, since it does all the work for us. The only thing that has to be done when Spoofax requests a reference lookup is a simple transformation: `node →<index−lookup> node`. The resulting definition has location information embedded into it which is used to navigate to the reference. If the lookup fails, this is propagated back to Spoofax and no blue hyperlink will appear on the node under the cursor.

Constraint Checking. Constraint checking rules are called by Spoofax after analysis on every AST node. If a constraint rule succeeds it will return the message and the node where the error marker should be put on.

```
constraint−error:
  node → (key, "Duplicate definition")
  where
    <nam−unique> node ;
    key  := <nam−key> node ;
    defs := <index−lookup−one> key ;
    <gt> (<length> defs, 1)
```

Fig. 24. Duplicate definitions constraint check

The duplicate definition constraint check that was shown earlier is defined in Figure 24. First `nam−unique` (generated for unique definitions by the NBL) is used to see if the node represents a unique definition; non-unique definition such as partial classes should not get duplicate definition error markers. The identifier is retrieved using `nam−key` and a lookup in the current scope is done with `index−lookup−one`. If more than one definition is found, the constraint check succeeds and an error marker is shown on the node.

Code Completion. When code completion is requested in Spoofax, a completion node is substituted at the place where the cursor is. For example, if we request code completion on `VarRef("a")`, it will be substituted by `VarRef(COMPLETION("a"))` to indicate that the user wants to complete this identifier. See Figure 25 for the code completion implementation. We first retrieve the completion node and name using `collect−one`. Completion proposals are gathered by `index−lookup−all−levels` since it can handle partial identifiers. Finally the retrieved proposals are converted to names by mapping `index−uri−name` over them.

7 Evaluation and Discussion

Our aim with this work has been to design high-level abstractions for name resolution applicable to a wide range of programming languages. In this section we discuss the limitations of our approach and evaluate its applicability to different languages and other language features than those covered in the preceding sections.

Limitations. There are two areas of possible limitations of NBL. One is in the provided abstraction, the other is in the implementation algorithm that supports it. As for the provided abstraction, as a definition language, NBL is inherently limited in the number of features it can support. While the feature space it

supports is extensive, ultimately there may always be language features or variations that are not supported. For these cases, the definition of NBL, written in Stratego, can be extended, or it is possible to escape NBL and extend an NBL specification using handwritten Stratego rules. As for the implementation algorithm, NBL's current implementation strategy relies on laziness, and does not provide much control over the traversal for the computation of names or types. In particular, sophisticated type inference schemes are not supported with the current algorithm. To implement such schemes, the algorithm would have to be extended, preferably in a way that maintains compatibility with the current NBL definition language.

Coverage. During the design and construction of NBL, we have performed a number of studies on languages and language features to determine the extent of the feature space that NBL would support. In this paper we highlighted many of the features by using C# as a running example, but other languages that we studied include a subset of general-purpose programming languages C, Java, and domain-specific languages WebDSL [10], the Hibernate Query Language (HQL), and Mobl [12]. We also applied our approach to the Java Bytecode stack machine language using the Jasmin [17] syntax.

For our studies we used earlier prototypes of NBL, which led to the design as it is now. Notable features that we studied and support in NBL are partial classes, inheritance, visibility, lexical scoping, imports, type-based name resolution, and overloading; all of which have been discussed in Sect. 4. In addition, we studied aspect-oriented programming with intertype declarations and pointcuts, file-based scopes in C, and other features. Our design has also been influenced by past language definitions, such as SDF and Stratego. Altogether, it is fair to say that NBL supports a wide range of language features and extensive variability, but can only support the full range of possible programming languages by allowing language engineers to escape the abstraction. In future work, we would like to enhance the possibilities of extending NBL and design a better interface for escapes.

8 Related Work

We give an overview of other approaches for specifying and implementing name resolution. The main distinguishing feature of our approach is the use of linguistic abstractions for name bindings, thus hiding the low level details of writing name analysis implementations.

Symbol Tables. In classic compiler construction, symbol tables are used to associate identifiers with information about their definition sites. This typically includes type information. Symbol tables are commonly implemented using hash tables where the identifiers are indexed for fast lookup. Scoping of identifiers can be implemented in a number of ways; for example by using qualified identifiers as index, nesting symbol tables or destructively updating the table during program analysis. The type of symbol table influences the lookup strategy. When

using qualified identifiers the entire identifier can be looked up efficiently, but considering outer scopes requires multiple lookups. Nesting symbol tables always requires multiple lookups but is more memory efficient. When destructively updating the symbol table, lookups for visible variables are very efficient, but the symbol table is not available after program analysis. The index we use is a symbol table that uses qualified identifiers. We map qualified identifiers (URIs) to information such as definitions, types and uses.

Attribute Grammars. Attribute Grammars [16] (AGs) are a formal way of declaratively specifying and evaluating attributes for productions in formal grammars. Attribute values are associated with nodes and calculated in one or more tree traversals, where the order of computations is determined by dependencies between attributes.

Eli provides an attribute grammar specification language for modular and reusable attribute computations [13]. Abstract, language-independent computations can be reused in many languages by letting symbols from a concrete language inherit these computations. For example, computations `Range`, `IdDef`, and `IdUse` would calculate a scope, definitions, and references. A method definition can then inherit from *Range* and *IdDef*, because it defines a function and opens a scope. A method call inherits from *IdUse* because it references a function. These abstract computations are reflected by naming concepts of NBL and the underlying generic resolution algorithm. However, NBL is less expressive, more domain-specific. Where Eli can be used to specify general (and reusable) computations on trees, NBL is restricted to name binding concepts, helping to understand and specify name bindings more easily.

Silver [26] is an extensible attribute grammar specification language which can be extended with general-purpose and domain-specific features. Typical examples are auto-copying, pattern matching, collection attributes, and support for data-flow analysis. However, name analysis is mostly done the traditional way; an environment with bindings is passed down the tree using inherited properties.

Reference Attribute Grammars (RAGs) extend AGs by introducing attributes that can reference nodes. This substantially simplifies name resolution implementations. JastAdd [7] is a meta-compilation system for generating language processors relying on RAGs and object orientation. It also supports parametrized attributes to act as functions where the value depends on the given parameters. A typical name resolution as seen in [5,7,2] is implemented in lookup attributes parameterised by an identifier of use sites, such as variable references. All nodes that can have a variable reference as a child node, such as a method body, then have to provide an equation for performing the lookup. These equations implement scoping and ordering using Java code. JastAdd implementations have much more low level details than NBL declarations. This provides flexibility, but entails overhead on encoding and requires decoding for understanding. For example, scopes for certain program elements are encoded within a set of equations, usually implemented by early or late returns.

Visibility Predicates. CADET [20] is a notation for predicates and functions over abstract syntax tree nodes. Similar to attribute grammar formalisms, it

allows to specify general computations in trees but lacks reusable concepts for name binding. Poetsch-Heffter proposes dedicated name binding predicates [21], which can be translated into efficient name resolution functions [22]. In contrast to NBL, scopes are expressed in terms of start and end points and multi-file analyses are not supported.

Dynamic Rewrite Rules. In term rewriting, an environment passing style does not compose well with generic traversals. As an alternative, Stratego allows rewrite rules to create dynamic rewrite rules at run-time [3]. The generated rules can access variables available from their definition context. Rules generated within a rule scope are automatically retracted at the end of that scope. Hemel et al. [11] describe idioms for applying dynamic rules and generic traversals for composing definitions of name analysis, type analysis, and transformations without explicitly staging them into different phases. Our current work builds on the same principles, but applies an external index and provides a specialized language for name binding declarations.

Name analysis with scoped dynamic rules is based on consistent renaming, where all names in a program are renamed such that they are unequal to all other names that do not correspond to the same definition site. Instead of changing the names directly in the tree, annotations can be added which ensure uniqueness. This way, the abstract syntax tree remains the same modulo annotations. Furthermore, unscoped dynamic rewrite rules can be used for persistent mappings [14].

Textual Language Workbenches. Xtext [6] is a framework for developing textual software languages. The Xtext Grammar Language is used to specify abstract and concrete syntax, but also name bindings by using cross-references in the grammar. Use sites are then automatically resolved by a simplistic resolution algorithm. Scoping or visibility cannot be defined in the Grammar Language, but have to be implemented in Java with help of a scoping API with some default resolvers. For example field access, method calls, and block scopes would all need custom Java implementations. Only package imports have special support and can be specified directly in the Grammar Language. Common constraint checks such as duplicate definitions, use before definition, and unused definitions also have to be specified manually. This increases the amount of boilerplate code that has to be rewritten for every language.

In contrast to Xtext's Grammar Language, NBL definitions are separated from syntax definitions in Spoofax. This separation allows us to specify more advanced name binding concepts without cluttering the grammar with these concepts. It also preserves language modularity. When syntax definitions are reused in different contexts, different name bindings can be defined for these contexts, without changing the grammar. From an infrastructure perspective, Spoofax and Xtext work similarly, using a global index to store summaries of files and URIs to identify program elements.

EMFText [8] is another framework for developing textual software languages. Like Xtext, it is based on the Eclipse Modeling Framework [23] and relies on

metamodels to capture the abstract syntax of a language. While in Xtext this metamodel is generated from a concrete syntax definition, EMFText takes the opposite approach and generates a default syntax definition based on the UML Human-Usable Textual Notation [18] from the metamodel. Language designers can then customize the syntax definition by adding their own grammar rules.

In the default setup, reference resolution needs to be implemented in Java. Only simple cases are supported by default implementations [9]. JastEMF [4] allows to specify the semantics of EMF metamodels using JastAdd RAGs by integrating generated code from JastAdd and EMF.

References

1. Standard ECMA-334 C# language specification, 4th edn (2006)
2. Åkesson, J., Ekman, T., Hedin, G.: Implementation of a Modelica compiler using JastAdd attribute grammars. Science of Computer Programming 75(1-2), 21–38 (2010)
3. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fundamenta Informaticae 69(1-2), 123–178 (2006)
4. Bürger, C., Karol, S., Wende, C., Aßmann, U.: Reference Attribute Grammars for Metamodel Semantics. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 22–41. Springer, Heidelberg (2011)
5. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V. (eds.) Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, pp. 1–18. ACM (2007)
6. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Int. Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, pp. 307–309. ACM (2010)
7. Hedin, G.: An Introductory Tutorial on JastAdd Attribute Grammars. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 166–200. Springer, Heidelberg (2011)
8. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009)
9. Heidenreich, F., Johannes, J., Reimann, J., Seifert, M., Wende, C., Werner, C., Wilke, C., A§mann, U.: Model-driven modernisation of java programs with jamopp. In: Joint Proceedings of MDSM 2011 and SQM 2011. CEUR Workshop Proceedings, pp. 8–11 (March 2011)
10. Hemel, Z., Groenewegen, D.M., Kats, L.C.L., Visser, E.: Static consistency checking of web applications with WebDSL. Journal of Symbolic Computation 46(2), 150–182 (2011)
11. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. Software and Systems Modeling 9(3), 375–402 (2010)
12. Hemel, Z., Visser, E.: Declaratively programming the mobile web with mobl. In: Fisher, K., Lopes, C.V. (eds.) 2011 Int. Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2011, pp. 695–712. ACM (2011)

13. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Inf. 31(7), 601–627 (1994)
14. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 444–463. ACM (2010)
15. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 918–932. ACM (2010)
16. Knuth, D.E.: Semantics of context-free languages. Theory Comput. Syst. 2(2), 127–145 (1968)
17. Meyer, J., Downing, T.: Java Virtual Machine. O Reilly (1997)
18. Object Management Group: Human Usable Textual Notation Specification (2004)
19. Object Management Group: Object Constraint Language, 2.3.1 edn. (2012)
20. Odersky, M.: Defining context-dependent syntax without using contexts. Transactions on Programming Languages and Systems 15(3), 535–562 (1993)
21. Poetzsch-Heffter, A.: Logic-based specification of visibility rules. In: PLILP, pp. 63–74 (1991)
22. Poetzsch-Heffter, A.: Implementing High-Level Identification Specifications. In: Pfahler, P., Kastens, U. (eds.) CC 1992. LNCS, vol. 641, pp. 59–65. Springer, Heidelberg (1992)
23. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2009)
24. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, University of Amsterdam (September 1997)
25. Visser, E.: Program Transformation with Stratego/XT. In: Lengauer, C., Batory, D., Blum, A., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)
26. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Science of Computer Programming 75(1-2), 39–54 (2010)