The Spoofax Name Binding Language

Gabriël D. P. Konat, Vlad A. Vergu, Lennart C. L. Kats, Guido H. Wachsmuth, Eelco Visser
Delft University of Technology, The Netherlands
g.d.p.konat@student.tudelft.nl, {v.a.vergu, l.c.l.kats, g.h.wachsmuth, e.visser}@tudelft.nl

Abstract

In textual software languages, names are used to identify program elements such as variables, methods, and classes. Name analysis algorithms resolve names in order to establish references between definitions and uses of names. In this poster, we present the Spoofax Name Binding Language (NBL), a declarative meta-language for the specification of name binding and scope rules, which departs from the programmatic encodings of name binding provided by regular approaches. NBL aspires to become the universal language for name binding, which can be used next to BNF definitions in reference manuals, as well as serve the generation of implementations.

Categories and Subject Descriptors D.2.1 [*Requirements-/Specifications*]: Languages; D.3.2 [*Language Classifications*]: Very high-level languages

Keywords name binding, name resolution, declarative, meta-language, Spoofax

1. Introduction

Name binding is concerned with the relation between definitions and references through identifiers in textual software languages, including scope rules that govern these relations. Classical approaches to name binding provide definitions in terms of programmatic encodings that carry environments through tree traversals. Attempts at abstractions such as attribute grammars (3; 4) or dynamic rewrite rules (2) reduce the overhead of such programmatic encodings, but are still algorithmic in nature. Our goal is a *declarative* domain-specific language for name binding that can be used to explain the binding rules of a language *and* and from which an efficient name resolution algorithm can be automatically derived, much like grammar formalisms (EBNF) abstract from the programmatic encoding of parsers.

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

In this poster, we present the Spoofax Name Binding Language (NBL) (6), a declarative meta-language for the specification of name binding in terms of namespaces, definitions, references, scopes, and imports. From definitions in NBL, a compiler generates a language-specific name resolution strategy in the Stratego rewriting language (1) 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. NBL is integrated in the Spoofax Language Workbench (5), but should be reusable in other language processing environments.

2. Name Binding and Scope Rules

We discuss the core concepts of NBL and illustrate their usage for a subset of C#.

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. Each class in a C# program defines the name of a class. Figure 1 defines the classes `Env`, `Expr`, `BinOp`, `Plus` and `Let`. Base class declarations are references to class definitions. For example, class `Plus` has a reference to its base class `BinOp`.

An NBL specification consists of a collection of rules of the form `pattern : clause*`, where `pattern` is an abstract tree (term) pattern and `clause*` is a list of name binding declarations about the language construct that matches with `pattern`. Figure 2 shows the NBL specification for name analysis of the C# subset. The first rule declares that a node matching the pattern `Class(x,_,_)` defines a class with name `x`. The second rule declares that the term pattern `Base(x)` is a reference to a class with name `x`. Thus, `: BinOp` is a reference to class `BinOp`.

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. The `Let` class in the example has a method *and* a field with name `eval`; methods and fields have their own namespace in our C# subset.

Scopes *Scopes* restrict the visibility of definitions. Scopes can be nested and name resolution typically looks for definitions from inner to outer scopes. The example includes three definitions for a method `eval`. These definitions are not distinguishable by their namespace `Method` and their name `x`, but, they are distinguishable by their scope, i.e. their containing class. The **`scopes`** `ns` clause in NBL declares a construct to be a scope for namespace `ns`. The `Class` rule in the NBL specification scopes all names in the `Field` and `Method` namspaces. Methods in turn scope local variables in the `Variable` namespace.

Imports An *import* introduces definitions from another scope into the current scope, either under the same name or under a new name. An import that imports all definitions can be *transitive*. Inheriting from a base class corresponds to (transitively) importing methods and fields from the base class into the super class. For example the `Plus` class imports the fields `l` and `r` from `BinOp`, `BinOp` imports `eval` from `Expr`. The second rule in the NBL specification declares that a base class reference to class `x` transitively imports all elements from the `Field` and `Method` namespaces of that class into the surrounding scope.

Types Types also play a role in name resolution. When calling a method `e.m(e*)`, the resolution of the method depends on the type of the target `e` and the types of the arguments `e*`. For example, `l.eval(env)` refers to the `eval` method of the `Expr` interface, since `l` has type `Expr` and `env` has type `Env`. In NBL, the **`of type`** `t` clause of a **`defines`** declaration indicates a type assignment. The **`where`** `e` **`has type`** `t` clause can be used to retrieve the type of an expression. For example, the rule for `Call` computes the types of the arguments and target of the method call.

Editor services Modern IDEs provide a wide range of editor services where name resolution plays a crucial role. Traditionally, each of these services is handcrafted for each language supported by the IDE. We automatically generate a name resolution algorithm and editor services from an NBL definition. Reference resolution, constraint checking and code completion are automatically generated from an NBL specification. Figure 3 shows an example of the generated constraint checking editor service; error markers are shown for unresolved references. An example of the generated code completion can be seen in Figure 4.

```
interface Env { Env add(string name, int val); }
interface Expr { int eval(Env env); }
class BinOp : Expr { Expr l; Expr r; }
class Plus : BinOp {
 int eval(Env env) {
  return l.eval(env) + r.eval(env);
} }
class Let : Expr {
 string name; Expr eval; Expr body;
 int eval(Env env) {
  return body.eval(env.add(name, eval.eval(env)));
} }
```

Figure 1. Expression evaluation classes in C# subset.

```
namespaces Class Method Field Variable
rules
 Class(x, _, _) :
  defines Class x of type Type(x)
  scopes Field, Method

 Base(x) :
  refers to Class x
  imports Field, Method from Class x {transitive}

 Method(t, x, p*, _) :
  defines Method x of type (t*, t)
  scopes Variable
  where p* has type t*

 Call(exp, x, a*) :
  refers to Method x of type (t*, _) in Class c
  where a* has type t*
  where exp has type Type(c)

 Field(t, x) :
  defines Field x of type t

 Param(t, x) :
  defines Variable x of type t

 VarRef(x) :
  refers to Variable x
  otherwise refers to Field x
```

Figure 2. NBL specification for a subset of C#.

```
18 class Let : Exp {
19  sting name;
20  Expr t;
21  Expr body;
22
23  int eval(Env env) {
24   Env newEnv = env.add(nam, t.eval(env));
25   return body.eval(newEnv);
26  }
27 }
```

Figure 3. Constraint checking for C# subset.

```
12  Expr r;
13  int eval(Env env) {
14   return l.e(env) + r.eval(env);
15  }
16 }
17
```

Figure 4. Code completion for C# subset.

References

1 M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *SCP*, 72(1-2):52–70, 2008. 1

2 M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *FUIN*, 69(1-2):123–178, 2006. 1

3 T. Ekman and G. Hedin. Modular name analysis for java using jastadd. In *GTTSE*, pages 422–436, 2006. 1

4 U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *ACTA*, 31(7):601–627, 1994. 1

5 L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, 2010. 1

6 G. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Language-parametric name resolution based on declarative name binding and scope rules. In *SLE*, 2013. 1