EDAN70 - Compiler Project

JastAdd library for name analysis

Daniel Forsman
PI10, Lund Institute of Technology, Sweden
atp10dfo@student.lu.se

Jakub Górski
D07, Lund Institute of Technology, Sweden
dt07jg8@student.lth.se

Abstract

During the process of constructing a compiler there are several steps that are crucial to the analysis of code. This paper will stage one of these steps, namely the development of a *name analysis library*. This will allow programmers to easily modify the name analysis, which will enable more versatile prototyping of compilers at the name analysis stage. By exploring the possibility for a *name analysis library* for JastAdd, an attempt to implement a generic name analysis library will be made, such that it supports several languages.

Keywords Name Analysis library, scoping, name binding, JastAdd.

Introduction

JastAdd is a system for generating language-based tools, such as compilers [3, p.1]. The subject of this paper will be a JastAdd library whose purpose is to simplify creation of name analysis. Where simplifying denotes reduction of required code size to implement the *name analysis* procedure. Because almost all programming languages are required to map variable usages to their respective declaration (name analysis) it is desirable to investigate the potential simplicity of a generic name analysis implementation.

Name analysis is conventionally implemented using attribute grammar in JastAdd, where certain nodes in the abstract syntax tree (AST) are decorated with *attributes* which denote certain properties. Such properties are implemented by utilizing Java's object-orientation and JastAdd code, which creates a symbiosis between attribute grammars and object-orientation.

By observing many different kinds of name analysis rules for compilers we conclude that there are many similarities within these rules in programming languages, such as declarations being used to initialize variables . The developer of the compiler often has to write similar code for different languages. This takes effort and time. Figure 1 shows how we would like to have it.

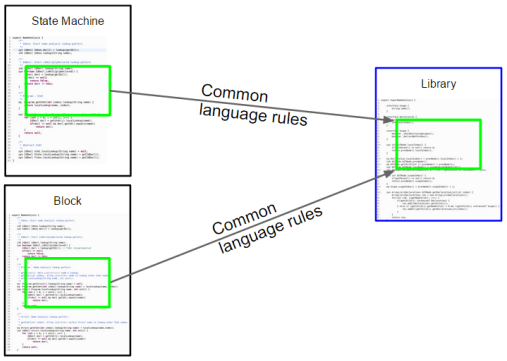

Figure 1. Illustration of how name analysis from the two languages State Machine and Block should be gathered into a library with common language rules

Alternatively, name analysis can be done using a generic tool in the form of a *library*, which is then used to implement name analysis, and thereby reducing code size. JastAdd's *interface* syntax allows attributes and JastAdd grammar [3, p.25] to be defined inside Java `Interface` classes. The goal is to write a library which uses the information gathered from abstract syntax tree classes to perform name analysis.

The result of our project is a .jrag file with some essential name analysis features: Name binding, declare before use, multiply-declared variables, and scoping. The programmer uses the library by implementing library-specific `interface` classes into the AST classes. Some of the interfaces have attributes which will be required to be set.

The library uses interfaces to represent classes that the user might implement. Consider the *declare before use* scenario. The user needs to create three classes: a symbol declaration class, a symbol usage class and a scope class. These should implement the library's class. The user can then specify attributes for how he wants the name analysis to be done. Here is a snippet of code showing an example code:

```
IdUse implements Usage;
IdDecl implements Declaration;
Program implements Scope;
```

```
syn boolean Program._declareBeforeUse() = true;
```

Our goal is to contain the name analysis library inside a .jrag file, which can analyze simple languages. This library contains Java `Interface` classes which the programmer extends the AST classes with. As the library matures more types of AST classes will gain the ability to be interfaced with the library, as well as a well defined library usage specification, however that will not be a subject for discussion in this paper.

The Name Analysis Library

The construction of the library is based on interfaces. The idea is to create your own AST classes and then ensure that those classes implement the appropriate interfaces which reside inside the library. For example if we would like to tell the library that our AST class `State` should be treated as a declaration , then the `State` class should implement the interface `Declaration`. Using the library the code would look like this:

```
State implements Declaration;
```

Some interfaces contain attributes, which are always required to be set. When this step above is done then the *library* and JastAdd will, if required, enforce certain attributes to be set by the user.

```
syn String State.name() = getStateID();
```

Where the `name()` variable is required by the library, and the `getStateID()` attribute returns the name of the declaration.

Upon implementing all appropriate interfaces the library will supply the user with a name analysis, by utilizing the information from the classes that are implementing library specific interfaces.

Implementation

This section is dedicated to explain how the library was implemented in this project and how it should be used.

Our implementation of the library relies on the fact that we can utilize interfaces and implement these interfaces into chosen classes and then use the attributes the library gives the user. When the user implements Declaration interface to the State AST class there will be attributes available to the user.

Declaration interface

The `Declaration` interface is the library specific interface used to represent a name declaration inside the library structure.

The library requires the user to bind the name attribute of the declaration to its appropriate AST class attribute, as it will be needed during the name analysis.

Usage interface

The `Usage` interface is the library specific interface used to represent a name use inside the library structure.

Just like with the `Declaration` interface the library enforces the user to specify the name of the variable that is being used. We need the user to do this so the library can find the declaration which corresponds to the usage. Example:

```
syn String IdUse.name() = getID();
```

The attribute gained from implementing this interface is `decl()`, which is an attribute that creates a reference between the node that declares the symbol and the node that uses the matching symbol. The `decl()` attribute is of the type `Declaration`.

Scope interface

The `Scope` interface is used to scope the name analysis library's internal AST into sections, whereas programmatically the `Scope` interface represents the beginning and end of a code block. Such scopes are utilized to trace the definition site of a symbol use, which in the context of the library would mean to group its AST into subtrees, which contain their respective `Declaration` and `Usage` classes. Here follows an example of a `Scope` in Java:

```
// Name Analysis .jrag file
Block implements Scope;

// Java code
int a = 0;
if(a) {  // this is a Block
  int b = 0;
}
```

This `Scope` interface enforces the user to specify two features. The first is whether the rule *declare before use* should be employed, `_declareBeforeUse()`. The second is checking for enclosing scopes, `_checkEnclosingScopes()`.

Lookup pattern

The lookup pattern has been implemented with accordance to what has been taught in the compiler course. The `Usage` interface has an attribute `decl()` which gets its value using the `lookup()` pattern. The `lookup()` pattern looks at its current scope and retrieves all declarations in the scope. If they match then a binding is created. If there is no match then the `_checkEnclosingScopes()` attribute is checked if the is true, if so then check the library will check enclosing scopes.

Enclosing scopes

The procedure to find enclosing scopes is simple. It looks at parents until it finds a parent node in the AST which is of instance `Scope`. At this point we have found the enclosing scope of the current scope.

If there are no further parents the top node has been reached and there are no more enclosing scopes.

Declare before use

In order for the library's internal data structure to be manageable a *local index* is given to each interface bound to an AST class. The *local index* is used to decide where in the program the declaration is placed, and numbers the AST classes. The *local index* is an attribute, which resides in the library, and is used by the library alone.

We check the *local index* of the symbol usage to see if it has a higher or equal value compared to each of the declarations it might be bound to. If the declare before use feature is set to false the local index will be ignored when looking for the declaration.

Multiply declared

To check if a symbol declaration already exists we give the `Declaration` interface a boolean attribute, `isMultiplyDeclared()`, and an attribute, `lookup()`. The `lookup` attribute works the same as the `lookup` for the `Usage` class. The attribute is either false or true depending on if the `lookup` finds a different declaration with the same name as the currently desired one.

This `lookup` assumes that the user is looking for symbols being declared before the current declaration.

Example

To test the result of the library we constructed an error handling aspect. This code is an example of how a developer would use the attributes in the library:

```
...

aspect ErrorContributions {
 IdDecl contributes error("Variable '" + getID()
   + "' is already declared.")
   when (isMultiplyDeclared())
   to Program.errors() for program();

 IdUse contributes error("Variable '" + getID()
   + "' is not declared.")
   when (decl() == null)
   to Program.errors() for program();
}
```

Error Handling language abstract grammar.

Here the attributes `isMultiyplyDeclared()` and `decl()` are used.

Internal library attributes

All the library attributes discussed above are useful. These are meant to be used. There are also internal library attributes that are used to construct them. These internal attributes can also be used by the user, but aren't meant to be used.

Evaluation

Deciding factors defining the efficiency of using the name analysis library is the amount of lines of code required to implement name analysis without the library, versus employing the name analysis library.

The name analysis library has been used to implement name analysis in three languages, which will be described in this section. The *State Machine* language is purely flat code, whereas the *Block* language supports scoping. The *Struct* language has been constructed in order to provide an estimate for its language analysis without using the library. The library, in its current state, cannot analyze this language however.

In Table 1 the results for lines of code are presented, where code comments have been excluded from the line count. The results show that the library works as intended.

	Implementation (LOC)	
Language	Manual	Library
State Machine	25 LOC	9 LOC
Block	29 LOC	9 LOC
Struct	81 LOC	-

Table 1. Required lines of code for implementing name analysis manually versus using the name analysis library counted using the tool CLOC. CLOC excludes comments and empty lines from the count.

State Machine language

The *State Machine* language has support for defining *states* and binding them by creating *transitions* between those two states. The language is defined by the following abstract grammar and library assisted implementation:

```
1 Program ::= Stmt*;
2
3 abstract Stmt;
4 State:Stmt ::= IdDecl;
5 Trans:Stmt ::= IdDecl From:IdUse To:IdUse;
6
7 IdDecl ::= <ID>;
8 IdUse ::= <ID>;
```

State Machine language abstract grammar.

```
aspect NameAnalysis {
 Program implements Scope;
 syn boolean Program._checkEnclosingScopes()
   = false;
 syn boolean Program ._declareBeforeUse() = true;

 IdUse implements Usage;
 IdDecl implements Declaration;
 syn String IdUse.name() = getID();
 syn String IdDecl.dname() = getID();
}
```
State Machine library assisted name analysis.

The library assisted name analysis defines scoping rules with the _checkEnclosingScopes() attribute, which in this case is set to false due to *state machine* language consisting of only one scope. The lack of scoping can be noted in the example *State Machine* code below:

```
state S1;
state S2;
trans a:S1->S2;
```
State Machine language example code.

Block language

The *Block* language is essentially the *State Machine* language, but with the additional ability have sever scopes. Thereby the _checkEnclosingScopes() attribute is set to true, which enables analysis through enclosing scopes at the name analysis stage. The Block language is specified by the following abstract grammar, and has the following library assisted implementation:

```
Program ::= Block;

abstract Stmt;
Block:Stmt ::= Stmt*;
State:Stmt ::= IdDecl;
Trans:Stmt ::= IdDecl From:IdUse To:IdUse;

IdDecl ::= <ID>;
IdUse ::= <ID>;
```
Block language abstract grammar.

```
aspect NameAnalysis {
 Block implements Scope;
 syn boolean Block._checkEnclosingScopes() = true;
 syn boolean Block._declareBeforeUse() = true;

 IdUse implements Usage;
 IdDecl implements Declaration;
 syn String IdUse.name() = getID();
 syn String IdDecl.dname() = getID();
}
```
Block library assisted name analysis.

If _declareBeforeUse() is set to false then all statements inside scopes Block are analyzed. Thereby making the following code example valid:

```
{
 state S1;
 {
  trans a:S1->S2;
  state S2;
 }
}
```
Block language example code.

Struct language

The *Struct* language has the ability to declare and access typed struct data structures. Additionally all variables are now typed, which requires *type analysis*. Currently the name analysis library doesn't do type analysis, and therefore doesn't meet the requirements to analyze this language.

Nonetheless, this language is described in this report because its name analysis has been implemented without name analysis library assistance, which results in data that can be seen in Table 1. Different potential directions of implementation will also be discussed.

The Struct language abstract syntax tree is defined by the following AST code:

```
Program ::= Struct* Stmt*;
Struct ::= StructType Stmt*;

abstract Stmt;
DeclarationStmt:Stmt ::= Type IdDecl;
AssignStmt:Stmt ::= UseExpr Expr;

abstract Expr;
abstract UseExpr:Expr;
IntLiteral:Expr ::= <NUMERAL>;
Dot:UseExpr ::= Left:UseExpr Right:IdUse;
IdUse:UseExpr ::= <ID>;

abstract Type;
IntType:Type ::= <INT>;
StructType:Type ::= IdDecl;

IdDecl ::= <ID>;
```
Struct language abstract grammar.

An example usage of the *Struct* language can be seen below:

```
struct B {
 int e;
}

struct A {
 B b;
 int d;
}

A a;
a.b.e = 5;
a.d = 44;
```

Struct language example code.

The functionality to analyze this language is not implemented in the library. However in order to extend the library with the ability to analyze this language new interfaces need to be coded into the library. Because one of those interfaces will be responsible for type analysis the library will require a lot more information regarding the AST simultaneously, such as both `decl()` and `type()` attributes. This poses a problem as combining interfaces, although possible, has not yet been evaluated and is not part of the library documentation.

Related Work

Name Binding Language NBL enables linguistic abstractions through declarative definitions of name binding and scope rules [5, p.2]. It is a declarative language which aims to generate name resolution rules defined in the *Stratego* language. Integration with *Spoofax Language Workbench*[5, p.14] is provided, with the possibility to extend existing language constraint rules through manually written *Stratego* rules[5, p.18].

The main difference between NBL and name analysis library is that we do not generate rules that are specified in a pre-existing constraint rule defining language, which in this case is *Spoofax*. Instead, the name analysis library already implements the most common language rules, then provides the programmer with interfaces to use them and attributes to moderate them.

Conclusion

The purpose of this project is to assist the implementation of the name analysis and explore the possibility of constructing a library for this purpose. We believe that the current library does this on a basic level. It implements only the fundamental name analysis. This could be useful for developers who want to skip implementing name analysis. Alternatively it can be used by students as an introduction to name analysis.

Future work would be to expand the library to cover more cases. We had a *Struct* language which we implemented by hand, but there was not enough time to implement it to the library. After implementing the functionality in the *Struct* language the next step might be classes, functions or type analysis.

One could always improve the library by adding more functions to the already existing interfaces e.g. scopes that are more specific, which could be allowing the programmer to, for example, define an attribute `scopeType()` which sets the type the Scope interface represents. Thereby gaining the ability to treat scopes differently, which could be useful when the programmer wants different library behavior for function statement scopes and if statement scopes.

There is also a need of changing the names of the attributes and interfaces to match a better standard. The library reserves all the names that are used in the library and forces the user to not use those names. This could create confusion when there is a bug saying that an attribute already exists when there are clearly no other attribute by that name. Since the library reserves the name for the attributes it uses we want the names to be consistent and not interfere too much with the users choice of name.

The ultimate goal with this project would be to have a library that can be used for any construction of a complete compiler for any language, and a name analysis library that can analyze any language.

Acknowledgments

Niklas Fors for supervising us during the project.

References

[1] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, August 2008.

[2] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, October 2007.

[3] Grel Hedin. An introductory tutorial on jastadd attribute grammars. In JooM. Fernandes, Ralf Lmmel, Joost Visser, and Joo Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 166–200. Springer Berlin Heidelberg, 2011.

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

[5] Gabril D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Grel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012.