Semantics driven disambiguation

Renaud Durlin

Technical Report n^o0709, June 13, 2008
revision 1464
BibTeX reference: durlin.07.seminar

An elegant approach to manage ambiguous grammars consists in using a Generalized LR (GLR) parser which will not produce a parse tree but a parse forest. An additional step, called disambiguation, occurring just after the parsing, is then necessary. The disambiguation process consists in analyzing the parse forest to choose the only good parse tree using semantics rules.

We use this approach in TRANSFORMERS with the Attribute Grammars (AGs) formalism. This report will present a comparison between this formalism and two other methods of disambiguation: the first one using *ASF+SDF* and the second one using *Stratego* language.

The goal of this comparison is to try to emphasize that AGs are well suited to solve the disambiguation problem. Another thing will be to find the weaknesses of this method compared to the two others for a possible improvement of the system used in TRANSFORMERS.

Une approche élégante pour gérer les grammaires ambiguës consiste à utiliser un parseur LR généralisé qui produira non pas un arbre mais une forêt de parse. Une étape supplémentaire, appelée désambiguisation, survenant juste après le parsing, est alors nécessaire. Celle-ci consiste à analyser cette forêt pour obtenir l'unique arbre valide correspondant à l'entrée en prenant en compte les règles de sémantiques contextuelles.

C'est cette approche qui a été retenue dans TRANSFORMERS avec le formalisme des grammaires attribuées. Le travail effectué présentera une comparaison entre ce formalisme et deux autres techniques de désambiguisation : la première à l'aide d'*ASF+SDF* et la deuxième à l'aide du langage *Stratego*.

Le but de cette comparaison sera double : montrer que les grammaires attribuées sont adaptées à ce problème et exhiber les faiblesses de celles-ci par rapport aux deux autres méthodes en vue d'une amélioration possible du système utilisé dans TRANSFORMERS.

Keywords
Transformers, context-free grammar, attribute grammar, Stratego, ASF+SDF, disambiguation, parsing, program transformation, term rewriting

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
renaud.durlin@lrde.epita.fr – http://www.lrde.epita.fr

Copying this document

Copyright © 2007 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just "Copying this document", no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Contents

1 Scannerless Generalized Parsing **6**
1.1 Context-free grammars . . . 6
1.2 Generalized Parsing . . . 6
1.3 Parse forests . . . 7
1.4 Scannerless Parsing . . . 7
1.5 Combining Scannerless Parsing and Generalized Parsing . . . 7

2 Disambiguation methods **8**
2.1 ASF – Algebraic Specification Formalism . . . 8
2.1.1 Algebraic Specification Formalism . . . 8
2.1.2 Syntax Definition Formalism . . . 8
2.1.3 The ASF+SDF Meta-environment . . . 9
2.2 Stratego – Strategies for Program Transformation . . . 9
2.2.1 Stratego Language . . . 9
2.2.2 Stratego/XT . . . 9
2.3 Attribute grammars . . . 10
2.3.1 Attribute grammars in Stratego/XT . . . 10

3 The Core language **11**
3.1 Declaration and use . . . 11
3.2 Scope . . . 12
3.3 Namespace . . . 12
3.4 Structs . . . 13
3.5 Typedef . . . 14
3.6 Using and Using namespace . . . 15
3.7 Template . . . 16

4 Disambiguation process **17**
4.1 Declaration and use . . . 17
4.2 Scope . . . 20
4.3 Namespace . . . 22
4.4 Structs . . . 24
4.5 Typedef . . . 25
4.6 Using and Using namespace . . . 26
4.7 Template . . . 28

5 Discussion **29**
5.1 Strengths 29
5.2 Weaknesses 30

6 Bibliography **32**

Introduction

Since the introduction of efficient deterministic parsing techniques, LR parsing became very often used in Computer Science and in Computational Linguistics. Tools based on deterministic parsing algorithms such as Lex & YACC (LALR) are considered adequate for dealing with almost all the modern programming languages.

The main problem is that grammars of languages such as COBOL, PL/1, FORTRAN, C++, etc. are not naturally LALR. Standard LR parsers cannot accommodate for the non-deterministic or even ambiguous nature of these languages.

GLR parsers operate on the same principle as LR parsers, but may be used with *all* Context-free grammars (CFGs), including ambiguous ones. An LR parser may reach a point where it cannot decide what action to take; a GLR parser doesn't care, simulating non-determinism and effectively taking all actions.

With GLR parsers, a parse tree can't be used because the parser may produce two or more different possible outputs. So a set of parse trees, called a parse forest, must be used. This parse forest encodes all the possible derivations for the input text. Obviously, the semantic rules of the language must be taken in consideration to select the one parse tree that corresponds to the input, because syntactic considerations are not sufficient. This *disambiguation* process takes place just after the parsing.

In this report we describe three different methods for this semantics *disambiguation*: term rewriting guided by algebraic specification (*ASF+SDF*), term rewriting using user-defined strategies (*Stratego*) and attribute grammars formalism (*sdf-attribute*). Moreover, we discuss about the strengths and weaknesses of each one.

Acknowledgments Thanks to Thomas Moulard, Samuel Charron and Nicolas Pierron.

Chapter 1

Scannerless Generalized Parsing

This chapter briefly describes the effects of using CFGs and scannerless generalized parsing in order to turn a possibly ambiguous input text to a parse forest.

1.1 Context-free grammars

In linguistics and computer science, a CFG is a formal grammar in which every production rule is of the form (1.1):

$$V \to w \tag{1.1}$$

where V is a non-terminal symbol and w is a string consisting of terminals and/or non-terminals. The term *context-free* expresses the fact that the non-terminal `V` can always be replaced by `w`, regardless of the context in which it occurs.

Context-free grammars exhibit some properties that are beneficial to language development:

- They are closed under union, unlike subclasses such as LALR. This enables to define modular grammars. The readability and maintainability of the grammars being developed is significantly increased.
- They don't impose restrictions on the grammar. In practice, this implies that there is absolutely no need to massage and obfuscate a grammar before its use.
- Ambiguous languages can be specified with a context-free grammar.

1.2 Generalized Parsing

Generalized parsing refers to a parsing method that yields all the possible representations for an ambiguous input. A popular approach to achieve this feature is GLR (Tomita (1985)). Given a context-free grammar and a possibly ambiguous input text, a GLR parser is able to generate all the trees that correspond to the input. This result is called a *parse forest*.

In this report, we use the Scannerless Generalized LR (SGLR) (Visser (1997)) generic parser. A parse table is generated from the Syntax Definition Formalism (SDF) context-free grammar. This table is given to the SGLR parser, along with the input text. Eventually, a parse forest is given as result.

1.3 Parse forests

In order to be reasonably space efficient, a parse forest is generally encoded in a single tree using ambiguity nodes. The children of such a node represent each possible derivation from this node. Moreover, maximal sub-tree sharing is used in order to avoid a massive duplication of nodes in memory. These two techniques permit a compact representation of parse forests.

The SGLR parser uses the ATerm format (Centrum voor Wiskunde en Informatica, 2004) to encode the parse forest. This format supports maximal term sharing, a compact binary representation, and term annotations.

1.4 Scannerless Parsing

Traditionally, syntax analysis is divided into a lexical scanner and a (*context-free*) parser. A scanner divides an input string consisting of characters into a string of tokens.

Although this architecture proves to be practical in many cases and is globally accepted as the standard solution for parser generation, it has some problematic limitations. Only few existing programming languages are designed to fit this architecture, since these languages generally have an ambiguous lexical syntax.

Scannerless Parsing tries to solve these limitations using only one formalism to express both the lexical and context-free syntax used to parse a language.

Advantages:

- Only a single meta-syntax is needed.
- Non regular lexical structure is handled easily.
- "Token classification" is not required (eliminating the need for "the lexer hack") when a lexical sequence may have different interpretations.
- The lexer-parser distinction is not required; languages without one are handled easily.

1.5 Combining Scannerless Parsing and Generalized Parsing

Syntax definitions in which lexical and context-free syntax are fully integrated do not usually fit in any restricted class of grammars required by deterministic parsing techniques because lexical syntax often requires arbitrary length lookahead. Therefore, scannerless parsing does not go well with deterministic parsing.

Generalized parsing techniques, on the other hand, can deal with arbitrary length lookahead. Using a generalized parsing technique solves the problem of lexical lookahead in scannerless parsing. However, it requires a solution for disambiguation of lexical ambiguities that are not resolved by the parsing context.

Chapter 2

Disambiguation methods

This chapter introduces the three different methods used for the comparison: ASF+SDF, Stratego and Attribute grammars.

2.1 ASF – Algebraic Specification Formalism

ASF+SDF is intended for the high-level, modular, description of the syntax and semantics of computer-based formal languages. It is the result of the marriage of two formalisms Algebraic Specification Language (ASF) and SDF.

2.1.1 Algebraic Specification Formalism

The Algebraic Specification Formalism (van den Brand et al., 2003) supports conditional rewrite rules and traversal functions using a user-defined (concrete) syntax. It allows the concise specification of program transformation, therefore it is suitable for semantics driven disambiguation, as described by van den Brand et al. (2003).

2.1.2 Syntax Definition Formalism

The Syntax Definition Formalism SDF is intended for the high-level description of grammars for programming languages, application languages, domain-specific languages, data formats and other computer-based formal languages.

SDF is a Syntax Definition Formalism with the following features:

- Modular syntax definition (parametrized modules, symbol renaming)
- Integrated lexical and context-free syntax
- Declarative disambiguation constructs (priorities, associativity, and more)
- Regular expression shorthands
- All non-circular context-free grammars allowed

2.1.3 The ASF+SDF Meta-environment

"The ASF+SDF Meta-Environment is a component-based language development environment. It is a complete system that can be used for many different purposes." The Meta-Environment consists of the following major components:

ToolBus A software application architecture that utilizes a scripting language based on process algebra to describe the communication between software tools.

ATerms Generic data representation, maximal sub-term sharing, automatic garbage collection.

MetaStudio Configurable GUI, structure editors, graph visualization.

ApiGen Generation of strongly typed APIs. ApiGen is a tool that generates C or Java code to implement tree-like data-structures.

PGEN & SGLR Parse-table generation and parsing.

ASFE & ASFC The ASF interpreter (ASFE) and the ASF compiler (ASFC). Efficient rewrite engines.

TIDE Generic debugging framework.

2.2 Stratego – Strategies for Program Transformation

Stratego/XT is a language and toolset for program transformation. The Stratego language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations, thus supporting the development of transformation components at a high level of abstraction.

The XT toolset offers a collection of extensible, reusable transformation tools, such as powerful parser and pretty-printer generators and grammar engineering tools. Stratego/XT supports the development of program transformation infrastructure, domain-specific languages, compilers, program generators, and a wide range of meta-programming tasks.

2.2.1 Stratego Language

The Stratego language focuses on term rewriting using user-defined strategies. It features conditional rewrite rules, generic traversal, scoped dynamic rules and supports term specification in concrete syntax. Refer to Visser (2004) for a description of the language.

2.2.2 Stratego/XT

The Stratego/XT bundle (de Jonge et al., 2001) regroups several tools that can be used together in order to build dedicated program transformation tools:

- the Stratego compiler and the Stratego Standard Library.
- a tool that turns SDF grammars into parsing tables,
- the SGLR generic parser,
- and GPP the Generic Pretty Printer (de Jonge, 2000).

2.3 Attribute grammars

Attribute grammars is a formalism introduced by Donald Knuth (Knuth, 1968) to specify the meaning of languages defined by a context free grammar.

Attribute grammars allow the embedding of attributes in the parse tree. These attributes can be computed using the attributes of other nodes. Those attributes that are computed using values from the parent node are *inherited*, whereas those using values from the child nodes are *synthesized*. An evaluator propagate all missing attributes and create a dependency graph.

2.3.1 Attribute grammars in Stratego/XT

Our implementation (David, 2004) embeds Stratego code in the SDF grammar in order to compute the values. Attributes are arbitrary terms, which can be trees, integers, strings, or even tables. This flexibility is useful because we need to handle various types of attributes, such as symbol tables. After the parsing, an evaluator computes the attributes until a fix point is reached.

Chapter 3

The Core language

For our comparison, an ambiguous grammar is needed. But C or C++ ones are too big and hard to disambiguate. So a small language called Core language (renamed Phenix when the language became bigger) was defined. An ambiguous grammar that generate this language was written in SDF.

The grammar exhibits some ambiguities introduced by the standard ISO C grammar (ISO/IEC, 1999), and replicated in the standard ISO C++ grammar (ISO/IEC, 2003).

3.1 Declaration and use

```
context-free syntax
  Id              -> Int
  Id              -> Float

  "int"           -> TypeDecl
  "float"         -> TypeDecl

  (Int | Float) -> TypeUse
  (Int | Float) -> TypeDecl

  TypeDecl Id   -> Decl
  TypeUse       -> Use

  Decl ";"      -> DeclStm
  Use  ";"      -> UseStm

  (DeclStm | UseStm)      -> Stm

  Stm*          -> Prog
```

Listing 3.1: Module Core

This part of the grammar (Listing 3.1) lets us declare *int* and *float*, using the `int` and `float`

keywords. Then we can use the identifiers declared this way. The problem comes from the fact that the grammar, being context-free, cannot make the difference between an *int* and a *float* when an identifier is used.

This ambiguity is not present in C or C++ but is useful because it looks like a kind of typage. And typage is needed to be able to disambiguate C++.

The previous grammar allows use to write something like this (Listing 3.2):

```
int foo;
float bar;
foo;
bar;
```

Listing 3.2: Core: example

In this example, `foo` is an *int* and `bar` is a *float*.

3.2 Scope

The grammar in Listing 3.3 imports the core grammar, and extends it with scoping. The main news is that a variable can be bound to another type in an inner scope.

```
imports
  Core

exports
  context-free syntax
    "{" Stm+ "}"   -> Stm
```

Listing 3.3: Module Scope

At this point, we can write something like this (Listing 3.4):

```
int foo;
{
  float foo;
  foo;
}
foo;
```

Listing 3.4: Scope: example

In this example, the first `foo` is a *float* but the second one is an *int*.

3.3 Namespace

Listing 3.5 adds namespaces (named scopes) to the grammar. This extension combines the two previous problems:

- When using the notation with "::", the grammar cannot make the difference between an *int* and a *float*.
- A declaration can hide a previous one. Outside of the namespace, the prior type must be retrieved.

```
imports
  Scope

exports
  context-free syntax
    "namespace" Id "{" Stm+ "}"  -> Stm

    Id              -> Namespace

    Namespace       -> NamespaceName

    NamespaceName "::" NamespaceName      -> NamespaceName

    NamespaceName "::" TypeUse            -> TypeUse
    NamespaceName "::" TypeDecl           -> TypeDecl
```

Listing 3.5: Module Namespace

Listing 3.6 shows what this new extension adds.

```
namespace A
{
  int foo;
  foo;
}
A::foo;
```

Listing 3.6: Namespace: example

3.4 Structs

Then Structs are added (Listing 3.7).

```
struct A { int foo; };
A::foo;
```

Listing 3.7: Struct: example

Note that all members inside a Struct are static, the notation "::" can be used with Structs too. This choice was made because it adds a new kind of ambiguity.

When the parser see something like in Listing 3.7, it must not only check if *foo* is an `int` or a `float` but also if *A* is a `namespace` of a `struct`.
The corresponding grammar is shown in Listing 3.8.

```
imports
  Namespace

exports
  context-free syntax
    "struct" Id "{" DeclStm* "}"   -> Decl

    Id            -> Struct

    Struct        -> NamespaceName

    Struct        -> TypeUse
    Struct        -> TypeDecl
```

Listing 3.8: Module Struct

3.5 Typedef

A `Typedef` declaration introduces a name that, within its scope, becomes a synonym for the given type.
The new grammar is shown in Listing 3.9 and an example in Listing 3.10.

```
imports
  Struct

exports
  context-free syntax
    "typedef" TypeDecl Id -> Decl

    Id                   -> Typedef

    Typedef              -> NamespaceName
    Typedef              -> TypeDecl
```

Listing 3.9: Module Typedef

```
typedef int t;
t x;
x;
```

Listing 3.10: Typedef: example

In order to be able to find the type of `x`, we must check to which type `t` is a synonym.

3.6 Using and Using namespace

A classic extension of namespaces is introduced by Listing 3.11. An `using namespace` gives access to all variables in a named scopes outside of it. And an `using` gives access to only one variable (the one specified in the `using`).

This extension adds a lot of work because it alters in depth the list of variables that are reachable.

```
1 imports
2   Typedef
3
4 exports
5   context-free syntax
6     "using" "namespace" NamespaceName ";" -> Stm
7
8     "using" TypeUse ";"                   -> Stm
```

Listing 3.11: Module Using

For example in Listing 3.12, after the `using namespace` at line 3, `foo1` and `foo2` are reachable. And after the `using` at line 4, only `bar` is reachable (not `baz`).

```
1 namespace A { int foo1; int foo2; }
2 namespace B { int bar; int baz; }
3 using namespace A;
4 using B::bar;
```

Listing 3.12: Using: example

3.7 Template

Finally, the last and hardest extension `Template` is described in Listing 3.13.

In computer programming, templates are a feature of the language that allow code to be written without consideration of the data type with which it will eventually be used.

This extension adds a simplified version of C++ template with only one parameter. Template specialization is also added.

```
imports
  Using

exports
  context-free syntax
    "template" "<" Id ">" "struct" Id "{" DeclStm* "}" -> Decl
    "template" "<" ">" "struct" Id "<" TypeDecl ">"
      "{" DeclStm* "}" -> Decl

    Id "<" TypeDecl ">"    -> Struct
```

Listing 3.13: Module Template

Listing 3.14 shows a small example of `Template`.

```
template <T>
struct S { T x; };

S<int>::x;

template <>
struct S<float> { float x; };

S<float>::x;
```

Listing 3.14: Template: example

Chapter 4

Disambiguation process

This chapter introduces the disambiguation process starting from the very basic construct to the hardest ones.

4.1 Declaration and use

Lets start with the first construct of our language (Listing 4.1).

```
1 int foo;
2 foo;
```

Listing 4.1: Declaration and Use

As already reported, the ambiguity is when foo is used. We must remember the declaration of foo to be able to find that foo is an *int*.

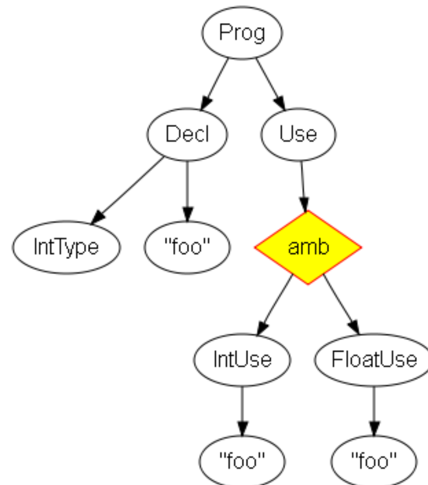

Figure 4.1: Declaration and Use

Figure 4.1 is the tree corresponding to the source in Listing 4.1. We can see the *amb* node under the *Use* node.

ASF+SDF In order to implement the disambiguation using the ASF+SDF Meta-Environment, we must extend the original SDF grammar with several rules to be able to write the ASF equations with concrete syntax.

We add three kinds of things:

- rules to specify the traversal.
- rules to define the environment table.
- and variables to enable concrete syntax in ASF equations.

We define a traversal strategy, which is a *transformer* (the tree can be modified during the traversal) and an *accumulator* (an environment is transmited during the traversal). Moreover, we specify the type of the traversal: `top-down` (try to visit current node and then traverse children) and `continue` (continue traversal even if an equation succeed). This strategy needs to be declared for each node that we intend to traverse.

In order to keep a context of what have been declared before, we use an environment table that contains a list of `Pair`.

The disambiguation module consists in a list of equations (Listing 4.2). The names between brackets are the names of the equations.

The first equation is used to declare an identifier. When the traversal strategy is applied to a "`int Id`" node (notice the concrete syntax), with a given environment, then the `Id` is added to the environment, associated with the `i` kind.

The disamb equations is used on an *Use* node. If an *int* is found in the environment table, the disamb equation is used and rewrite the *amb* node into an *int* node.

Similar equations are defined for *float* declaration and use.

```
1  equations
2    [env-i] disamb(int Id, [ P* ]) = <int Id, [ P* Id : i ]>
3
4    [disamb-i]
5      Id := i,
6      [ P*1 Id : i P*2 ] := env
7      ===========================================
8      disamb(amb(U*1, i, U*2), env) = <i, env>
```

Listing 4.2: ASF equations

Stratego In order to disambiguate our parse forest, we build a filter written in Stratego. This filter rewrites the parse forest into a parse tree, resolving the ambiguities using specified traversals and transformations. Moreover, dynamic rules are used to keep the necessary knowledge about the declared symbols.

Stratego is modular, just like SDF. The filter in Listing 4.3 imports libstratego-lib - the standard library. This Stratego module consists in a set of strategies. The main strategy, `core-disamb`, is the combination of other strategies:

```
module core-disamb

imports
  libstratego-lib
  Phenix

strategies
  decl = ?VarDecl(IntType(), x)
       ; rules(use:+ amb(as) -> t where
                <getfirst(?IntUse(x))> as => t)

  decl = ?VarDecl(FloatType(), x)
       ; rules(use:+ amb(as) -> t where
                <getfirst(?FloatUse(x))> as => t)

  disamb = decl <+ use <+ all(disamb)

  core-disamb = io-wrap(disamb)
```

Listing 4.3: Stratego: core-disamb

- `io-wrap` is a wrapper that handles the input and output of the term. This is the strategy that gives our binary the ability to read from standard input or from a file, write to standard output or to a file, and accept options to change the behavior of the filter.
- `all` basically calls a strategy on all sons of the current term.
- `decl` is the strategy we use to declare the identifiers. In fact, `decl` will create dynamic rules which are named `use`. The two `decl` strategies are similar, and they will be combined using non-deterministic choice in order to be called by the main strategy. This means that each of them will be applied until one of them succeed.

  The first part (`?VarDecl(IntType(), x)`) tries to match an *int* declaration. If it fails, the strategy also fails. Otherwise, a dynamic rule is created, which rewrites an ambiguous node (`amb(as)`) to a `IntUse` if one of the children of the ambiguous node is an *int* identifier with the same name. Thus, we keep a context without using a symbol table. When a type declaration is found, it is known that future ambiguities about an identifier with the same name need to be resolved to an int. Therefore, a dynamic rule is spawned at this moment.

Attribute grammars In order to filter ambiguous nodes using attribute grammars, we use a special *ok* attribute. This attribute is a Boolean value which represents the validity of a node. The nodes which are not *ok* are removed from the tree after the attribute processing is done. Thus, we just need to adjust the *ok* attributes for some nodes, depending of the symbol tables that we generate, in order to strip the invalid derivations.

The way attributes are embedded in the SDF grammar is by the means of the `attributes` annotation. Each attribute resides in a namespace. In Listing 4.4 we use the *disamb* namespace. There is a single value computed in this rule, which is the `lr_table_syn` attribute of the root node (`Decl`). The computation is written in Stratego. Basically, it adds (`Id.string`,

```
1  TypeDecl Id  -> Decl
2    {cons("VarDecl"),
3    attributes(disamb:
4      root.lr_table_syn := ![(Id.string, TypeDecl.type)
5                             | root.lr_table_in]
6    )}
```

Listing 4.4: Attribute annotation for declaration

`TypeDecl.type`) to the environment table. `Id.string` is the identifier name and `TypeDecl.type` is the type of the variable.

The suffix *_in* means inherited propagation and *_syn* means synthesized propagation. Therefore, this rule just add the identifier's name to the *inherited* symbol table, mapped to a `Type`, and puts it in the *synthesized* table.

In order to disambiguate the parse forest, we must first identify the rules that cause the ambiguities. In our example, we know that the rules that map an `Id` to an `Int` or a `Float` are ambiguous. Therefore, we annotate these rules with the special attribute `ok`, as seen in Listing 4.5.

```
1  Id            -> Int
2    {attributes(disamb:
3      root.ok := <lookup(Id.string)> root.lr_table_in => Int()
4    )}
```

Listing 4.5: Attribute annotation for usage

We need to decide whether the corresponding node is the good one or not. In fact, we need to know if the `Id` has been declared previously as a `Int` or not, and this information is located in the symbol table.

The code in Listing 4.5 gets the tuple corresponding to the key `Id.string` and checks if it's an `Int`.

4.2 Scope

Now, we extend the grammar to add the first extension: Scope (Listing 4.6) that gives us the tree in Figure 4.2.

```
1  int foo;
2  {
3    float foo;
4    foo;
5  }
6  foo;
```

Listing 4.6: Scope

We have the same ambiguity as before, when `foo` is used, we must check if it is an *int* or a

float. But the new thing is that we must not remember the declaration made inside a scope when we leave it.

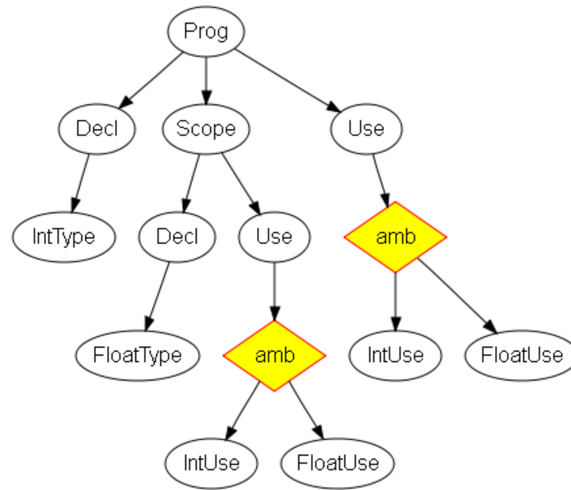

Figure 4.2: Scope

ASF+SDF In order to handle the disambiguation of scopes, we must use a "hack" in our grammar. We want to perform action when entering in a scope (store the environment) and another when leaving the scope (restore the environment). But ASF equations don't allow this. So we split the scope rule. Thus we can attach an equation to each rule.

To be able to restore the previous environment table when leaving the scope, we need to modify what we store. We add a list of environments in addition to the list of `Pair`.

Listing 4.7 shows the two associated equations. The first one stores the current environment `env` into the environment list. And the second, restores it. You can see the powerful lists matching: the environment is deconstruct automatically with `env env*`.

```
1  [enter-scope]
2    [ P*   env* ] := env,
3    env’  := [ P* env env* ]
4    ==========================================
5    disamb(S, env)  = <S, env’>
6
7  [end-scope]
8    disamb(E, [ P*  env env* ]) = <E, env’> when env’  := env
```

Listing 4.7: ASF equations for scopes

Stratego Listing 4.8 shows the Stratego code for disambiguating Scopes. As we use dynamic rules for disambiguation and not symbol table, we also use a special Stratego construct here: *scoped dynamic rules*.

First we check if we are on a scope (`?Scope(_)`). If this is the case, we call `all(disamb)` to traverse all sons of the current term. But it's not a simple call, we scope the rule *use*, with the special construction "`{|`" and "`|}`". So *all* dynamic rules *use* created inside this strategy will be discarded at the end of the strategy.

```
module scope-disamb

strategies
  enter-scope = ?Scope(_) ; {| use: all(disamb) |}
```

Listing 4.8: Stratego: scope-disamb

Attribute grammars Scopes are very easy to disambiguate with AGs. We don't want to propagate the table modified inside the scope leaving it. So we just need to synthesize the inherited table as shown in Listing 4.9.

```
"{" Stm+ "}"  -> Stm
  {cons("Scope"),
  attributes(disamb:
    root.lr_table_syn := !root.lr_table_in
  )}
```

Listing 4.9: Table copy for scopes

4.3 Namespace

We add the next extension to the grammar: Namespace (Listing 4.10 and the corresponding tree in Figure 4.3).

```
namespace A
{
  int x;
}
A::x;
```

Listing 4.10: Namespace

New information needs to be stored to be able to find variables inside namespaces. We must keep up-to-date something (a term, an attribute, ... depending on the method) which stores in which namespace we are to change the way we store variables and the way we lookup into the environment table.

ASF+SDF The same thing as for scopes is made for namespaces: split the rule in two: one equation when entering in a namespace and another one when leaving the namespace.

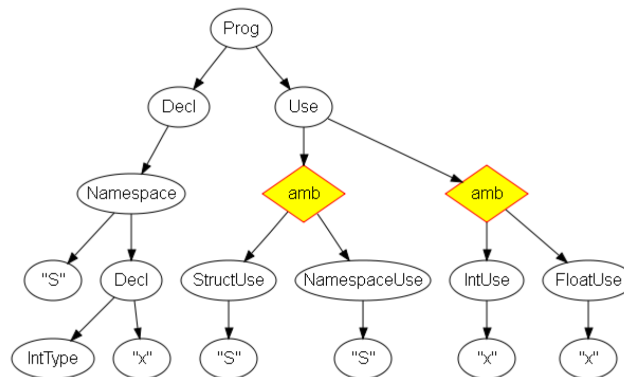

Figure 4.3: Namespace

The environment table must be changed another time to add new information, the current namespace.

The equations (Listing 4.11) aren't very complex. Append the namespace name at the end of the list (notice the concrete syntax to take the name) for the first one and remove it for the second.

```
1 [enter-ns]
2   disamb(namespace Id { S+ }, [ P* env* Ns* ]) =
3               <namespace Id { S+ }, [ P* env* Ns* Id ]>
4
5 [end-ns]
6   disamb(Ens, [ P*  env* Ns* Ns ]) = <Ens, [ P* env* Ns* ]>
```

Listing 4.11: ASF equations for namespaces

Stratego As for ASF+SDF, this extension is very intrusive because we must update all the previous strategies so now they take a term as parameter (|*ns*). It's the current namespace name used for namespace disambiguation.

In order to disambiguate *namespaces*, we must update this term as shown in Listing 4.12. First, we check that we are on a namespace (`?Namespace(n, _)`). Then we concatenate the namespace names and we use the strategy *all* to traverse the tree.

```
1 module namespace-disamb
2
3 strategies
4   enter-namespace(|ns) = ?Namespace(n, _)
5                         ; all(disamb(|[ns | n]))
```

Listing 4.12: Stratego : namespace-disamb

Attribute grammars We use a new attribute *ns* to store the current namespace name. We must update the previous code that stores and retrieves data from the environment table to use this new attribute.

For each production rules, we need to update the value of the attribute (see Listing 4.13). When we enter in a namespace, the new namespace name became the concatenation of the name of the new namespace (*Id.string*) and the current namespace name (*root.lr_ns_in*).

```
1  "namespace" Id "{" stm:Stm+ "}"  -> Stm
2    {cons("Namespace"),
3    attributes(disamb:
4      stm.lr_ns_in    := ![Id.string | root.lr_ns_in]
5    )}
```

Listing 4.13: Attribute annotation for namespaces

4.4 Structs

The next extension is Structs. An example is shown in Listing 4.14 and the corresponding tree in Figure 4.4.

```
1  struct S
2  {
3    int x;
4  };
5  S::x;
```

Listing 4.14: Struct

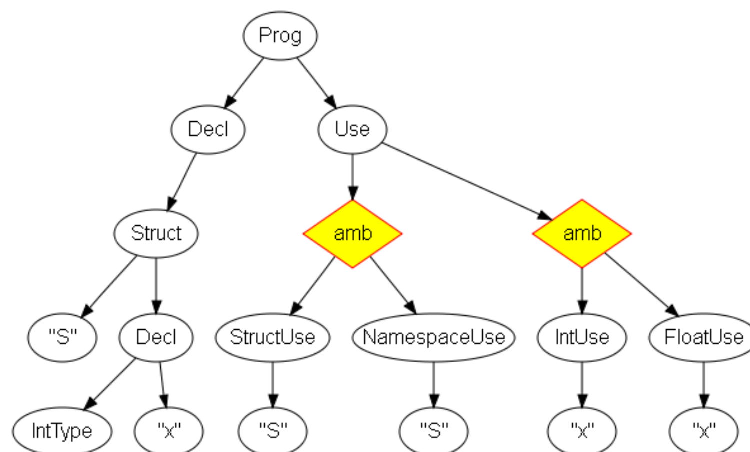

Figure 4.4: Struct

As for Namespace, we must know in which Struct we are and in which Struct a variable has been declared.

In fact, the problem is the same as disambiguating Namespace. We will use the same thing to store the current namespace or the current scope.

But this extension adds a new ambiguity, we must not only check if a variable is an `int` or a `float` but also if the identifier before the "`::`" is a `namespace` of a `struct`.

ASF+SDF The equations (Listing 4.15) show how the disambiguation is made with ASF+SDF. Like for namespaces, these equations append the namespace name at the end of the list (notice the concrete syntax to take the name) for the first one and remove it for the second.

```
1  [enter-st]
2    disamb(struct Id { S+ };, [ P* env* Ns* ]) =
3                <struct Id { S+ };, [ P* env* Ns* Id ]>
4
5  [end-st]
6    disamb(Est, [ P*  env* Ns* Ns ]) = <Est, [ P* env* Ns* ]>
```

Listing 4.15: ASF equations for Struct

Stratego As for ASF+SDF, this extension is very intrusive because we must update all the previous strategies so now they take a term as parameter (|*ns*): the current namespace name, used for namespace disambiguation.

In order to disambiguate *structs*, the term added for namespace disambiguation is re-used (see Listing 4.16). First, we check that we are on a struct (`?Struct(n, _)`. Then we concatenate the namespace names and we use the strategy *all* to traverse the tree.

```
1  module struct-disamb
2
3  strategies
4    enter-struct(|ns)  = ?Struct(n, _)
5                       ; all(disamb(|[ns | n]))
```

Listing 4.16: Stratego : struct-disamb

Attribute grammars For each production rule, we need to update the value of the `ns` attribute (see Listing 4.17). When we enter in a struct, the new namespace name become the concatenation of the name of the new struct (*Id.string*) and the current namespace name (*root.lr_ns_in*).

4.5 Typedef

A `Typedef` declaration introduces a name that, within its scope, becomes a synonym for the given type.

An example is shown in Listing 4.18 and the corresponding tree in Figure 4.5.

```
1  "struct" Id "{" stm:DeclStm* "}"  -> Decl
2    {cons("Namespace"),
3    attributes(disamb:
4      stm.lr_ns_in   := ![Id.string | root.lr_ns_in]
5    )}
```

Listing 4.17: Attribute annotation for structs

```
1  typedef int t;
2  t x;
3  x;
```

Listing 4.18: Typedef

In order to be able to find the type of `x`, we must check to which type `t` is a synonym.

This extension and following has not yet been done with ASF+SDF.

Stratego Listing 4.19 shows how the disambiguation is made with Stratego.
If we are on a typedef (`?Typedef(type, x)`), two new dynamic rules are created: the first one is used when we want to know the type of a variable. This rule returns the tuple with the type of the variable and the name of the variable.
The second rule is used when the ambiguous node is rewrite into the good sub-tree.

```
1  module typedef-disamb
2
3  strategies
4    typedef(|ns) = ?Typedef(type, x)
5          ; rules(get-type:+ amb(as) -> (type, t) where
6                <getfirst(?TypedefUse(x))> as => t)
7          ; rules(use:+ amb(as) -> t where
8                <getfirst(?TypedefUse(x))> as => t)
```

Listing 4.19: Stratego: typedef-disamb

Attribute grammars With AGs, the disambiguation process uses the environment table to store all the information needed to disambiguate `Typedef` (see Listing 4.20).
The constructor *Typedef()* is used to remember the data's type. The two parameters are used to store the current namespace name (this attribute also contains the name of the variable) and the type associated to the typedef.

4.6 Using and Using namespace

A `Using namespace` imports all variables in a named scopes outside of it. And a `Using` imports only one variable (the one specified in the `Using`).

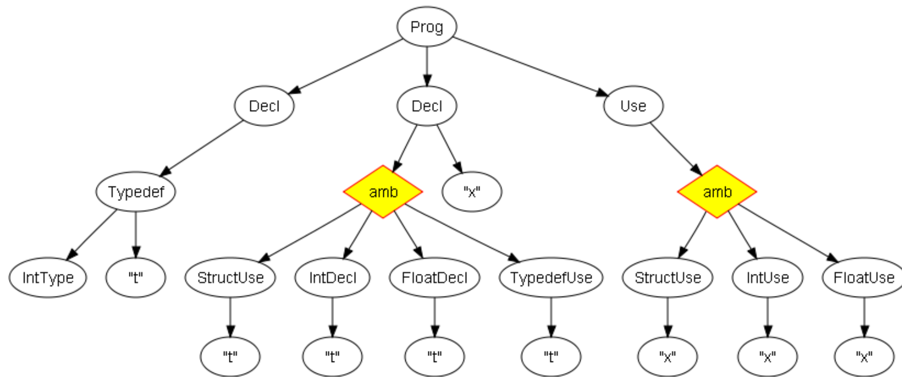

Figure 4.5: Typedef

```
"typedef" TypeDecl Id -> Decl
  {cons("Typedef"),
  attributes(disamb:
    root.lr_table_syn :=
      ![Typedef(TypeDecl.lr_ns_syn, TypeDecl.type)
       | TypeDecl.lr_table_syn]
  )}
```

Listing 4.20: Attribute annotation for typedef

An example is shown in Listing 4.21 and the corresponding tree in Figure 4.6.

Starting on this extension, the disambiguation is made only with AGs.

```
namespace A
{
  int x;
}
using namespace A;
x;
```

Listing 4.21: Using

This extension adds a lot of work because it alters in depth the list of variables that are reachable. This is a very intrusive extension because a lot of changes are needed.

To be able to know if a variable is reachable, the *ns* attribute doesn't store enough information. Another attribute is needed to store a list of reachable namespace. This attribute is called *access_ns*.

We need to add and keep up-to-date this attribute in all the rules seen previously. For example when we see a *namespace* for the first time, we initialized the attribute with a single list: the current namespace name.

An *Using namespace* extend the attribute adding a new reachable namespace into this list.

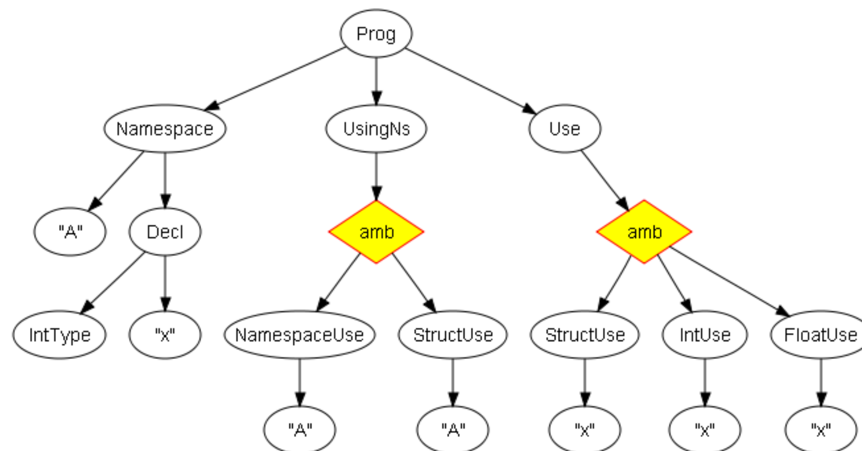

Figure 4.6: Using namespace

When we check if a variable is reachable, we no longer use the *ns* attribute. All namespaces stored inside the *access_ns* attribute are successively used to search the variable.

4.7 Template

In computer programming, templates are a feature of the language that allows code to be written without consideration of the data type with which it will eventually be used.

This extension adds a simplified version of C++ template with only one parameter. Template specialization is also added.

An example is shown in Listing 4.22.

```
1 template <T>
2 struct S { T x; };
3 S<int>::x;
```

Listing 4.22: Template

As the type is not known at the declaration, the sub-tree corresponding to the template cannot be disambiguated right now. But this tree may be needed in the future if the template is used. So we store the tree inside the environment table.

When the template is instantiated, we get enough information to be able to disambiguate the template sub-tree. The type of the parameter is added into the environment table and then the attribute evaluator is called with the template sub-tree.

The attributes are computed and the environment table is filled with the variables corresponding to the instantiation.

The hardest thing is that the disambiguation process must be delayed until all the needed information (the type of the parameter) is known.

Chapter 5

Discussion

5.1 Strengths

ASF+SDF

- Using equations to describe term rewriting leads to a clean formalism.
- Simple traversals have been added to this formalism, in order to concisely specify a transformation system.
- Tight coupling with concrete syntax.
- Support functional and algebraic specification.
- Fast and scalable underlying term rewriting engines.

Stratego

- Stratego code is very expressive.
- Complex traversals can be expressed very easily using the various combination operators and user-defined strategies.
- Additional flexibility comes with the concept of scoped dynamic rules, which allows to have a global context without explicitly carrying it everywhere.

Attribute grammars

- Attribute grammars are conceptually elegant, being a combination of declarativeness at the rule level and imperativeness at the attribute level. The rules are pretty much independent from each other.
- Attribute grammars are naturally extended, just like the SDF grammar. Since the disambiguation code is broken down at the rule level, adding new rules tends to fit well in the existing code.
- Using the Stratego language in order to compute the attributes brings several good features, such as the flexibility of the ATerm format, the whole Stratego standard library, and more generally the expressiveness of the language.

5.2 Weaknesses

ASF+SDF

- No strategies like in Stratego.
- No higher-order function.
- Complicated traversals are not trivial in ASF. A lot of equations have to be written, since the traversal must be explicitly described.
- Understanding a whole disambiguation module require a detailed reading of the equations, both in the module as well as in the imported ones. Since every rewrite rule is potentially executable on any node, there can be hidden interactions between some equations, leading to cycles or unwanted effects. Those are not too hard to debug, thanks to the execution trace, but they definitely do not appear at first sight.

Stratego The Stratego modules are not directly linked to the grammar it operates on. If the grammar is extended, care must be taken to make sure the Stratego code can handle it. Thus, code refactoring is sometimes necessary in order to extend the Stratego module. However, as we have seen in our example, when the code is well written, extension is be painless.

Attribute grammars

- Separation of concerns is hard to achieve. Attaching all the attribute processing with their rules means that different processing can't be easily separated or disabled. Our implementation does provide namespaces, which is an improvement concerning this problem, but this is not a full scale solution.
- Attribute grammars tend to clutter the grammar. The code for the attribute computation is somewhat mixed with the rule declarations and the other annotations. When the code gets long and complicated, the grammar becomes less readable.

Conclusion

As a conclusion, we have seen how to perform semantic disambiguation on a toy example. Three different approaches have been explored:

- Term rewriting using the Algebraic Specification Formalism (ASF), which is purely declarative.
- Term rewriting using the paradigm of strategies (Stratego), which is mostly imperative (functional style).
- Attribute grammars, using Stratego code to compute the attributes, thus mixing declarativeness and imperativeness.

Based on our previous experience with semantic disambiguation on the pretty big grammars of standard C/C++ (Anisko et al., 2003), it seems that attribute grammars is the formalism that fits best for this task. However, the work needs to be continued to see how the different paradigms deal with real-world languages constructs that cause bigger problems.

Chapter 6

Bibliography

Anisko, R., David, V., and Vasseur, C. (2003). Transformers: a C++ program transformation framework. Technical Report 0310, LRDE.

van den Brand, M., Klusener, S., Moonen, L., and Vinju, J. J. (2003). Generalized parsing and term rewriting: Semantics driven disambiguation. volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Centrum voor Wiskunde en Informatica (2004). The ATerm Library. http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary.

David, V. (2004). Attribute grammars for C++ disambiguation. Technical Report 0405, LRDE.

de Jonge, M. (2000). A pretty-printer for every occasion. In Ferguson, I., Gray, J., and Scott, L., editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia.

de Jonge, M., Visser, E., and Visser, J. (2001). XT: A bundle of program transformation tools. In van den Brand, M. G. J. and Perigot, D., editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

ISO/IEC (1999). ISO/IEC 9899:1999 (E). *Programming languages - C*.

ISO/IEC (2003). ISO/IEC 14882:2003 (E). *Programming languages - C++*.

Knuth, D. E. (1968). Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127–145.

Tomita, M. (1985). *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

van den Brand, M. G. J., Klint, P., and Vinju, J. J. (2003). Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190.

Visser, E. (1997). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

Visser, E. (2004). *Strategies for Program Transformation*. Draft available from http://www.stratego-language.org/Book.