Electronic Notes in Theoretical Computer Science 82 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 17 pages

Generalized Parsing and Term Rewriting: Semantics Driven Disambiguation

M.G.J. van den Brand[2,4], A.S. Klusener[3,4], L. Moonen[1], J.J. Vinju[1]

[1] *Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands*
Leon.Moonen@acm.org, Jurgen.Vinju@cwi.nl
[2] *LORIA-INRIA, 615 rue de Jardin Botanique, BP 101, F-54602 Villers-lès-Nancy Cedex, France*
[3] *Software Improvement Group, Kruislaan 419, NL-1098 VA Amsterdam, The Netherlands*
[4] *Vrije Universiteit, Division of Mathematics and Computer Science, De Boelelaan 1081A, NL-1081 HV Amsterdam, The Netherlands*
{markvdb,steven}@cs.vu.nl

Abstract

Generalized parsing technology provides the power and flexibility to attack real-world parsing applications. However, many programming languages have syntactical ambiguities that can only be solved using semantical analysis. In this paper we propose to apply the paradigm of term rewriting to filter ambiguities based on semantical information. We start with the definition of a representation of ambiguous derivations. Then we extend term rewriting with means to handle such derivations. Finally, we apply these tools to some real world examples, namely C and COBOL. The resulting architecture is simple and efficient as compared to semantic directed parsing.

1 Introduction

Generalized parsing is becoming more popular because it provides the power and flexibility to deal with real existing programming languages and domain specific languages [3,9]. It solves many problems that are common in more widely accepted technology based on LL and LR algorithms [2,13].

We start by briefly recalling the advantages of generalized parsing. It allows arbitrary context-free grammars instead of restricting grammars to classes like LL(k) or LALR(k). Due to this freedom, a grammar can better reflect the structure of a language. This structure can be expressed even better using modularity. Modularity is obtained because context-free grammars are closed under union, as opposed to the more restricted classes of grammars.

An obvious advantage of allowing arbitrary context-free grammars is that

the number of grammars accepted is bigger. It seems that real programming languages (e.g. Pascal, C, C++) do not fit in the more restricted classes at all. Without 'workarounds', such as semantic actions that have to be programmed by the user, off-the-shelf parsing technology based on the restricted classes can not be applied to such languages.

The main reason for real programming languages not fitting in the restricted classes is that they are ambiguous in one way or the other. Some grammars have simple conflicts that can be solved by using more look-ahead or by trying more alternative derivations in parallel. Generalized parsing offers exactly this functionality. Other grammars contain more serious ambiguities, which are all accepted as valid derivations. The result is that after parsing with a generalized parser we sometimes obtain a collection of derivations (a parse forest) instead of a single derivation (a parse tree).

1.1 Examples

Many examples of the more serious ambiguities can be found in existing programming languages. In this section we will discuss briefly a number of ambiguous constructs which are hard to solve given traditional parsing technology.

Typedefs in C In the C programming language certain identifiers can be parsed as either type identifiers or variable identifiers due to the fact that certain operators are overloaded:

```
{ Bool *b1; }
```

The above statement is either a statement expression multiplying the `Bool` and `b1` variables, or a declaration of a pointer variable `b1` to a `Bool`. The latter derivation is chosen by the C compiler only if `Bool` was declared to be a type using a `typedef` statement somewhere earlier in the program, otherwise the former derivation is chosen. Section 4 describes a solution for this problem via the technology presented in this paper.

Offside rule Some languages are designed to use indentation to indicate blocks of code. Indentation, or any other line-by-line oriented position information is obviously not properly expressible in any context-free grammar, but without it the syntax of such a language is ambiguous. The following quote explains the famous offside rule [19] from the users' perspective:

> "The southeast quadrant that just contains the phrase's first symbol must contain the entire phrase except possibly for bracketed sub-segments."

For example, the following two sentences in typical functional programming style illustrate an ambiguity:

```
a = b                                   a = b
    where b = d                             where b = d
          where d = 1       vs.                 where d = 1
          c = 2                                       c = 2
```

On the left-hand side, the variable `c` is meant to be part of the first `where` clause. Without interpretation of the layout of this example, `c` could just as well part of the second where clause, as depicted by the right-hand side.

There are several languages using some form of offside rule, among others, Haskell [12]. Each of these languages applies "the offside rule" in a different manner making a generic definition of the rule hard to formalize.

Nested dangling constructions in COBOL For C and Haskell we have shown ambiguities that can only be solved using context information. A similar problem exists for COBOL, however we will present a different type of ambiguity here that is based on complex nested statements. [1]

The following example resembles the infamous *dangling else* construction, but it is more complex due to the fact that more constructs are optional. Consider the following piece of COBOL code in which a nested `ADD` statement is shown:

```
0001 ADD A TO B
0002   SIZE ERROR
0003     ADD C TO D
0004       NOT SIZE ERROR
0005         CONTINUE
0006 .
```

The `SIZE ERROR` and `NOT SIZE ERROR` constructs are optional post-fixes of the `ADD` statement. They can be considered as a kind of exception handling. In order to understand what is going on we will present a tiny part of a COBOL grammar, which is based on [18]:

```
Add-stat              ::= Add-stat-simple Size-error-phrases
Size-error-phrases    ::= Size-error-stats? Not-size-error-stats?
Size-error-stats      ::= "SIZE" "ERROR" Statement-list
Not-size-error-stats ::= "NOT" "SIZE" "ERROR" Statement-list
Statement-list        ::= Statement*
```

The above grammar shows that the COBOL language design does not provide explicit scope delimiters for some deeply nested `Statement-lists`. The result is that in our example term, the `NOT SIZE ERROR` can be either part of the `ADD`-statement on line `0001` or `0003`. The period on line `0006` closes both statements.

The COBOL definition does not have an offside rule. Instead it states that in such cases the "dangling" phrase should always be taken with the innermost

[1] There are many versions of the COBOL programming language. In this paper we limit ourselves to IBM VS COBOL II.

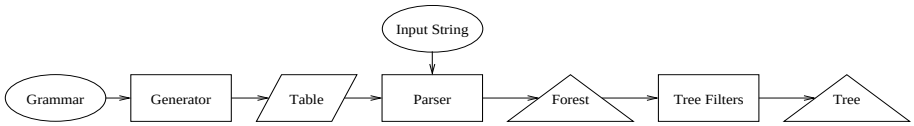

Fig. 1. General parsing and disambiguation architecture

construct, which is in our case the `ADD`-statement on line `0003`. There are 16 of such ambiguities in the COBOL definition. Some of them interact because different constructs might be nested.

In some implementations of LL and LR algorithms such dangling cases are implicitly resolved by a heuristic that *always* chooses the deepest derivation. In this paper we describe a more declarative and maintainable solution that does not rely on such a heuristic.

Discussion The above examples indicate that all kinds of conventions or computed information might have been used in a language design in order to disambiguate its syntax. This information can be derivable from the original input sentence, or from any other source.

Generalized parsing is robust against any grammatical ambiguity. So, we can express the *syntax* of ambiguous programming languages as descriptive context-free grammars. Still, in the end there must be only one parse tree. The structure of this parse tree should faithfully reflect the *semantics* of the programming language. In this paper we will fill this gap between syntax and semantics, by specifying how to disambiguate a parse forest.

1.2 Related work on filtering

The parsing and disambiguation architecture used in this paper was proposed earlier by [15] and [24]. An overview is shown in Figure 1. This architecture clearly allows a separation of concerns between syntax definition and disambiguation. The disambiguation process was formalized using the general notion of a *filter*, quoted from [15]:

> "A filter F for a context-free grammar G is a function from sets of parse trees for G to sets of parse trees for G, where the number of resulting parse trees is equal to or less than the original number."

This rather general definition allows for all kinds of filters and all kinds of implementation methods. In [9] several declarative disambiguation notions were added to context-free grammars. Based on these declarations several filter functions were designed that discard parse trees on either *lexical* or *simple structural* arguments. Because of their computational simplicity several of the filters could be implemented early in the parsing process. This was also possible because these filters were based on *common* ambiguity concepts in language design.

In this paper we target more *complex structural* parse tree selections and selections based on *non-local* information. More important, we aim for *lan-*

guage specific disambiguations, as opposed to the more reusable disambiguation notions. Such filters naturally fit in at the back-end of the architecture, just before other semantics based tools will start their job. In fact, they can be considered part of the semantic analysis.

Wagner and Graham [24] discuss the concept of disambiguation filters including an appropriate parse forest formalism, but without presenting a formalism for implementing disambiguation filters. This paper complements their work by describing a simple formalism based on term rewriting which allows the user to express semantics-guided disambiguation. Furthermore, we give a 'proof of concept' by applying it to real programming languages.

The notion of *semantics/attribute directed parsing* [1,5] also aims to resolve grammatical ambiguities that can be solved using semantical information. However, the approach is completely different. In case of semantics directed parsing the parser is extended to deal with derived semantic information and directly influence the parsing process. Both in the specification of a language and in the implementation of the technology syntax and semantics become intertwined. We choose a different strategy by clearly separating syntax and semantics. The resulting technology is better maintainable and the resulting language specifications also benefit from this separation of concerns. For example, we could replace the implementation of the generalized parser without affecting the other parts in the architecture [2].

1.3 Filtering using term rewriting

Given the architecture described, the task at hand is to find a practical language for implementing language specific disambiguation *filters*. The functionality of every disambiguation filter is similar, it analyzes and prunes parse trees in a forest. It does this by inspecting the structure of sub-trees in the parse forest and/or by using any kind of context information.

An important requirement for every disambiguation filter is that *it may never construct a parse forest that is ill-formed with respect to the grammar of a language*. This requirement ensures that the grammar of a language remains a valid description of the parse forest and thus a valuable source of documentation [14], even after the execution of any disambiguation filters.

The paradigm of *term rewriting* satisfies all above mentioned requirements. It is designed to deal with terms (read trees); to analyze their structure and change them in a descriptive and efficient manner. Term rewriting provides exactly the primitives needed for filtering parse forests. Many implementations of term rewriting also ensure well-formedness of terms with respect to the underlying grammar (a so-called *signature*). Term rewriting provides a solid basis for describing disambiguation filters that are concise and descriptive.

[2] If the parse forest representation remains the same.

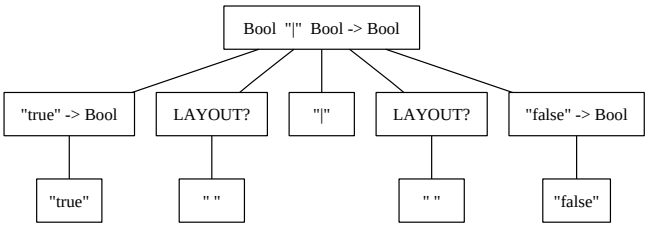

Fig. 2. A parse tree for "`true | false`".

1.4 Plan of the paper

In the rest of this paper we give the implementation details of disambiguation filters with term rewriting. In Section 2, we give a description of the parse tree formalism we use. Section 3 briefly describes term rewriting basics before we extend it with the ability to deal with forests instead of trees. In Section 4 we give a number of examples with details to show that it works for real programming languages. In Section 5, we discuss how our techniques can be applied to other paradigms and describe our future plans. We present our conclusions in the final section.

2 Parse Forest Representation

Based on a grammar, the parser derives valuable information about how a sentence has to be structured. However, the parser should also preserve any information that might be needed for disambiguation later on. The most obvious place to store all this information is in the syntax tree.

Furthermore, we need a practical representation of the alternative derivations that are the result of grammatical ambiguity. Ambiguities should be represented in such a way that the location of an ambiguous sub-sentence in the input can be pinpointed easily. Just listing all alternative parse trees for a complete sentence is thus not acceptable.

In this section we describe an appropriate parse tree formalism, called AsFix. A more elaborate description of AsFix can be found in [21]. We will briefly discuss its implementation in order to understand the space and time efficiency of the tools processing these parse trees.

AsFix is a very simple formalism. An AsFix tree contains all original characters of the input, including white-space and comments. This means that the *exact* original sentence can be reconstructed from its parse tree in a very straightforward manner. Furthermore, an AsFix tree contains a complete description of all grammar rules that were used to construct it. In other words, all valuable information present in the syntax definition and the input sentence is easily accessible via the parse tree.

Two small examples illustrate the basic idea. Figure 2 shows a parse tree of the sentence "`true | false`". Figure 3 shows a parse tree of the ambiguous input sentence "`true | false | true`". We have left out the white-space

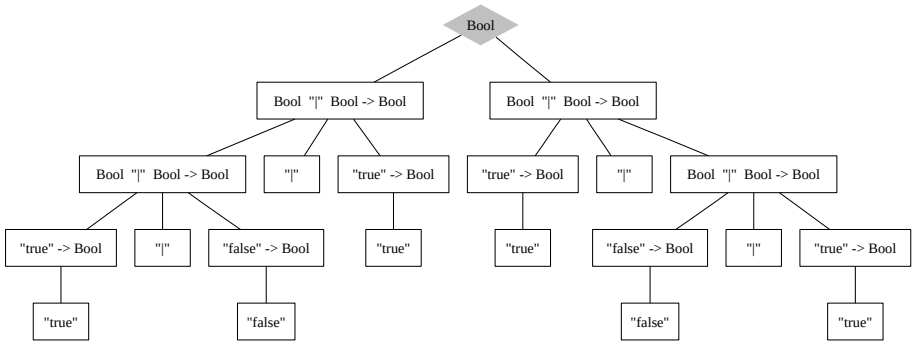

Fig. 3. A parse forest with two ambiguous derivations for "`true | false | true`".

nodes in latter picture for the sake of presentation. The diamond represents an *ambiguity* node which indicates that several derivation are possible for a certain sub-sentence. The following grammar (in SDF [11]) was used to parse these two sentences:

```
context-free syntax
  "true"        -> Bool
  "false"       -> Bool
  Bool "|" Bool -> Bool
```

The implementation of AsFix is based on the ATerm library [7]. An AsFix parse tree is an ordinary ATerm, and can be manipulated as such by all utilities offered by the ATerm library. The ATerm library is a library that implements the generic data type ATerm. ATerms are a simple tree-like data structure designed for representing all kinds of trees. The characteristics of the ATerm library are maximal sub-term sharing and automatic garbage collection.

The maximal sharing property is important for AsFix for two reasons. Firstly, the parse trees are completely self-contained and do not depend on a separate grammar definition. It is clear that this way of representing parse trees implies much redundancy. Maximal sharing prevents unnecessary occupation of memory caused by this redundancy. Secondly, for highly ambiguous languages parse forests can grow quite big. The compact representation using ambiguity nodes helps, but there is still a lot of redundancy between alternative parse trees. Again, the ATerm library ensures that these trees can be stored in an minimal amount of memory. To illustrate, Figure 4 shows the parse forest of Figure 3 but now with full sharing.

3 Extending Term Rewriting

In this section we explain how a parse tree formalism like AsFix can be connected to term rewriting. This connection allows us to use term rewriting directly to specify disambiguation filters. The important novelty is the lightweight technique that is applied to be able to deal with ambiguities. After explaining it we present a small example to illustrate the style of specification

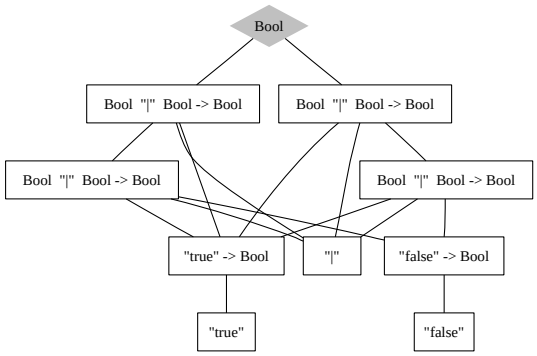

Fig. 4. A parse forest with maximal sharing.

used for defining disambiguation filters. More elaborate examples are given in Section 4. We start by briefly and informally describing the basics of term rewriting.

3.1 What is term rewriting?

In short, a Term Rewriting System (TRS) is the combination of a *signature* and a collection of *rewrite rules*. The signature defines the prefix *terms* that are to be rewritten according to the rewrite rules. We refer to [16] for a detailed description of term rewriting.

Signature A many-sorted signature defines all possible terms that can occur in the rewrite rules and the term that is to be rewritten. Usually a signature is extended with a collection of *variables*, which are needed for specifying the rewrite rules. The following is an example signature, with constants (nullary functions), function applications, and a variable definition:

```
signature
  true           -> Bool
  false          -> Bool
  or(Bool,Bool)  -> Bool
variables
  B -> Bool
```

This signature allows ground terms like: "`or(true,or(false,true))`". *Ground* means containing no variables. Terms containing variables are called *open* terms.

Rules A rewrite rule is a pair of such terms `T1 = T2`. Both `T1` and `T2` may be open terms, but `T2` may not contain variables that do not occur in `T1`. Furthermore, `T1` may not be a single variable. A ground term is called a *redex* when it *matches* a left-hand side of any rule. Matching means equality modulo occurrences of variables. The result of a match is a mapping that assigns the appropriate sub-terms to the variable names. A *reduct* can be constructed by taking the right-hand side of the rule and substituting the variables names using the constructed mapping. Replacing the original redex by the reduct is

called a *reduction*. Below we give an example of a set of rewrite rules:

```
rules
  or(true, B) = true
  or(false, B) = B
```

The redex "`or(false, false)`" matches the second rule, yielding the binding of `B` to the value `false`. The reduct is `false` after substitution of `false` for `B` in the right-hand side.

In most implementations of term rewriting, the rewrite rules are guaranteed to be *sort-preserving*. This implies that the application of any rewrite rule to a term will always yield a new term that is well-formed with respect to the signature.

Normalization Given a ground term and a set of rewrite rules, the purpose of a rewrite rule interpreter is to find all possible redexes in a larger term and applying all possible reductions. Rewriting stops when no more redexes can be found. We say that the term is then in *normal form*.

A frequently used strategy to find redexes is the *innermost* strategy. Starting at the leafs of the tree the rewriting engine will try to find reducable expressions and rewrite them. For example, "`or(true,or(false,true))`" can be normalized to `true` by applying the above rewrite rules in an innermost way.

Associative matching Lists are a frequently occurring data structure in term rewriting. Therefore, we allow the `*` symbol to represent repetition in a signature:

```
signature
  set(ELEM*) -> SET
variables
  E       -> ELEM
  Es[123] -> ELEM*
```

The argument of the `set` operator is a list of `ELEM` items. By using *list variables* [3], we can now write rewrite rules over lists. The following examples removes all double elements in a `SET`:

```
rules
  set(Es1,E,Es2,E,Es3) = set(Es1,E,Es2,Es3)
```

A list variable may bind any number of elements, so left-hand sides that contain list variables may match a redex in multiple ways. One possible choice of semantics is to take the first match that is successful and apply the reduction immediately.

[3] `Es[123]` declares three variables, `Es1`, `Es2` and `Es3`, using character class notation.

3.2 Rewriting parse trees

Grammars as signatures The first step is to exploit the obvious similarities between signatures and context-free grammars. We replace the classical prefix signatures by arbitrary context-free grammars in a TRS. There are three immediate consequences. The non-terminals of a grammar are the sorts. The grammar rules are the function symbols. Terms are valid parse trees over the grammar. Of course, the parse trees can be obtained automatically by parsing input sentences using the user-defined grammar.

Rules in concrete syntax If we want to rewrite parse trees, the left-hand side and right-hand side of rewrite rules should be parse trees as well. We use the same parser to construct these parse trees.[4] In order to parse the variables occurring in the rules, the grammar has to be extended with some variables as well.

Using grammars as signatures and having rules in concrete syntax, the TRS for the `or` can now be written as:

```
context-free syntax
  "true"        -> Bool
  "false"       -> Bool
  Bool "|" Bool -> Bool
variables
  "B"           -> Bool
rules
  true  | B = true
  false | B = B
```

A formalism like this allows us to use term rewriting to analyse anything that can be expressed using an *unambiguous* context-free grammar.

Brackets In order to be able to explicitly express the structure of terms and to be able to express rewrite rules unambiguously, the notion of bracket rules is introduced. The following grammar adds a bracket production to the booleans:

```
context-free syntax
  "(" Bool ")" -> Bool {bracket}
```

Bracket productions may only be *sort-preserving*. This allows that applications of bracket productions can be removed from a parse tree without destroying the well-formedness of the tree. The result of this removal is a parse tree with the structure that the user intended, but without the explicit brackets.

3.3 Rewriting parse forests

The step from rewriting parse trees to rewriting parse forests is a small one. If we want to use term rewriting to design disambiguation filters, we want to

[4] We implicitly extend the user-defined grammar with syntax rules for the rewrite rule syntax, e.g. `Bool "=" Bool -> Rule`, is added to parse any rewrite rule for booleans.

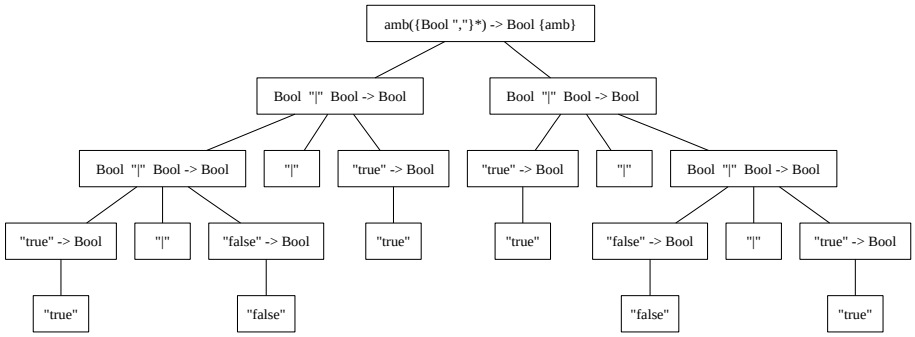

Fig. 5. Translating an ambiguity node to an ambiguity constructor.

be able to address ambiguities *explicitly* in a TRS.

Extending the signature The ambiguity nodes that exist in a parse forest are made visible to a TRS by augmenting the signature automatically with a new function symbol for every sort in the signature. The new function symbols are referred to as *ambiguity constructors*. For example, the following ambiguity constructor is added for `Bool` expressions:

```
context-free syntax
  amb({Bool ","}*) -> Bool {amb}
```

Each ambiguity constructor has a comma separated list of children. These children represent the original ambiguous derivations for a certain sub-sentence.

Preprocessing before rewriting Just before the normalization process begins we translate each `amb` node in the parse forest of an input term to an application of an ambiguity constructor as described in the previous paragraph. The result of this translation is a *single* parse tree, representing a parse forest, that is completely well-formed with respect to the augmented signature of the TRS.

Figure 5 depicts the result of this translation process. It shows how the parse forest from Figure 3, containing an ambiguity node, is translated to a parse tree with an explicit ambiguity constructor. Due to the reserved production attribute {`amb`} the translation back is a trivial step. This is done after the normalization process is finished and there are still ambiguities left. This step is necessary in order to make the normal form completely well-formed with respect to the original grammar.

Since the extended signature allows empty ambiguity clusters, e.g. `amb()`, the final translation can sometimes not be made. In this case we return an error message similar to a parse error. An empty ambiguity constructor can thus be used to indicate that a term is semantically incorrect.

Rewrite rules Using the ambiguity constructors, we can now define rewrite rules which process ambiguity clusters. The following example specifies the removal of right-associative application of the `"|"` operator. This is performed by the first rewrite rule

```
variables
  "Bs"[12] -> Bool*
  "B"[123] -> Bool
rules
  amb(Bs1, B1 | (B2 | B3), Bs2) = amb(Bs1, Bs2)
  amb(B1) = B1
```

The second rule transforms an ambiguity cluster containing exactly one tree into an ordinary tree. Using innermost normalization, the above rewrite rules will rewrite a parse forest of the ambiguous term "`true | false | true`" to a parse tree that represents "`(true | false) | true`".

The following features of term rewriting are relevant for this example. Concrete syntax allows specification of the functionality directly in recognizable syntax. The use of brackets is essential to disambiguate the left-hand side of the first rewrite rule. Associative matching is used to locate the filtered derivation directly without explicit list traversal. Finally, the innermost strategy automatically takes care of executing this rewrite rule at the correct location in the parse tree.

4 Practical Experiences

The above ideas have been implemented in ASF+SDF [10], which is the combination of the syntax definition formalism SDF and the term rewriting language ASF. ASF+SDF specifications look almost like the last examples in the previous section.

C typedefs To test the concept of rewriting parse forests, we developed a small specification that filters one of the ambiguities in the C language. We started from an ambiguous syntax definition of the C language.

The ambiguity in question was discussed in the introduction. Depending on the existence of a `typedef` declaration an identifier is either interpreted as a type name or as a variable name. The following specification shows how a C `CompoundStatement` is disambiguated. We have constructed an environment containing all declared types. If the first `Statement` after a list of `Declarations` is a multiplication of an identifier that is declared to be a type, the corresponding sub-tree is removed. Variable definitions have been left out for the sake of brevity.

```
context-free syntax
  "types" "[[" Identifier* "]]"  -> Env
  filter(CompoundStatement, Env) -> CompoundStatement
equations
[]Env = types[[Ids1 Id Ids2]]
  ==========================
  filter(amb(CSs1,{Decls Id * Expr;Stats},CSs2),Env) = amb(CSs1,CSs2)
```

Note the use of concrete C syntax in this example. The filter function searches

and removes ambiguous block-statements where the first statement uses an identifier as a variable which was declared earlier as a type. Similar rules are added for every part of the C syntax where an ambiguity is caused by the overlap between type identifiers and variable identifiers. This amounts to about a dozen rules. Together they both solve the ambiguities and *document* exactly where our C grammar is ambiguous.

The offside rule in Sasl We have experimented with Sasl [20], a functional programming language, to implement a filter using the offside rule. The following function uses *column* number information that is stored in the parse forest to detect whether a certain parse tree is offside.

```
[] Col = get-start-column(Expr),
   minimal(Expr,Col) < Col = true
   ==============================
   is-offside(NameList = Expr) = offside
```

An expression is offside when a sub-expression is found to the left of the beginning of the expression. The function `minimal` (not shown here) computes the minimal column number of all sub-expressions.

After all offside expressions have been identified, the following function can be used to propagate nested `offside` tags upward in the tree:

```
[] propagate(Expr WHERE offside) = offside
[] propagate(NameList = offside) = offside
[] propagate(offside WHERE Defs) = offside
```

Next, the following two rules are used to remove the offside expressions:

```
[] amb(Expr*1, offside, Expr*2) = amb(Expr*1,Expr*2)
[] amb(Expr) = Expr
```

Note that if no expressions are offside, the ambiguity might remain. Rules must be added that choose the deepest derivation. We have left out these rules here for the sake of brevity because they are similar to the next COBOL example.

Complex nested dangling COBOL statements The nested dangling constructs in COBOL can be filtered using a simple specification. There is no context information involved, just a simple structural analysis. The following rewrite rules filter the derivations where the dangling block of code was not assigned to the correct branch:

```
equations
[] amb(ASs1, AddStatSimple1
              SIZE ERROR Stats1 AddStatSimple2
              NOT SIZE ERROR Stats2,
       ASs2) = amb(ASs1, ASs2)
```

	Size (bytes)	Lines	Parse time (seconds)	Number of ambiguities	Filter time (seconds)
Smallest file	10,454	158	16.57	0	0.46
Largest file	203,504	3,020	36.66	1	10.55
File with most ambiguities	140,256	2,082	28.21	8	7.21
Largest file without ambiguities	124,127	1,844	26.61	0	8.79
Totals of all files	5,179,711	79,667	1818.67	125	259.25
Averages	59,537	916	20.90	1.44	2.98

Table 1
Some figures on parsing and filtering performance.

The variable `AddStatSimple2` terminates the nested statement list. In the rule, the `NOT SIZE ERROR` is therefore assigned to the outer `AddStatSimple1` statement instead of the inner `AddStatSimple2`.

This is exactly the alternative that is *not* wanted, so the rule removes it from the forest. We have defined similar disambiguation rules for each of the 16 problematic constructions.

Performance To provide some insight in the computational complexity of the above COBOL disambiguation we provide some performance measures. We used a rewrite rule *interpreter* for these measurements. Compiling these rules with the ASF-compiler [6] would lead to a performance gain of at least a factor 100. However, it is easier to adapt the ASF interpreter when prototyping new language features in ASF. In Table 1 we compare the parsing time to time the rewriter used for filtering. The figures are based on a test system of 87 real COBOL sources, with an average file size of almost 1,000 lines of code.

The parse times include reading the COBOL programs and the parse table from disk, which takes approximately 15 seconds, and the construction and writing to disk of the resulting parse forests. The parse table for this full COBOL grammar is really huge, it consists of 28,855 states, 79,081 actions, and 730,390 gotos. The corresponding grammar has about 1,000 productions. It was derived from the original Reference Manual of IBM via the technique described in [18].

The time to execute this set of disambiguation filters for COBOL is proportional to the size of the files and not to the number of ambiguities. The computation visits every node in the parse tree once without doing extensive computations.

5 Discussion

Object-oriented programming We demonstrated the concept of semantic filters via rewriting. An important question is what is needed to apply the same idea in a more general setting, for instance using Java or an Attribute Grammar formalism. We will formulate the requirements and needed steps as a recipe:

(i) It is necessary to have a parse forest or abstract syntax forest representation which has ambiguity clusters. The amount and type of information stored in the trees influences the expressiveness of the disambiguation filters directly.

(ii) The ambiguity nodes should be made addressable by the user.

(iii) It is necessary to create a mapping from the output of the parser to this parse forest representation. This mapping should preserve or derive as much information as possible from the output of the parser.

(iv) If possible, it is preferable to guarantee that the output of a filter is well-formed with respect to the grammar.

Programming filters becomes a lot simpler if there exists a practical application programming interface (API) to access the information stored in the parse forest representation. JJForester [17] is a Java class hierarchy generator that provides a mapping from AsFix to a typed abstract syntax tree in Java that is ready for the Visitor design pattern. The current implementation of JJForester also incorporates a representation for abstract syntax *forests*.

Strategic programming In our description of term rewriting we have not addressed the notion of first class rewriting strategies that is present in languages like Stratego [22] and Elan [4]. Rewriting strategies allow the specification writer to explicitly control the application of rewrite rules to terms, as opposed to using the standard innermost evaluation strategy. Ambiguity constructors can be combined seamlessly with rewriting strategies, allowing disambiguation rules to be applied under a certain user-defined strategy. Recently both Elan and Stratego started to use SDF to implement concrete syntax too, [8] and [23], respectively.

6 Conclusions

Starting from the notion of generalized parsing we have presented a solution for one of its implications: the ability to accept ambiguous programming languages. Term rewriting can be extended in a simple manner to filter parse forests. Specifying filters by explicitly addressing ambiguity clusters is now as simple as writing ordinary rewrite rules.

The resulting architecture provides a nice separation of concerns and declarative mechanisms for describing syntax and disambiguation of real programming languages.

Practical experience shows that writing filters in term rewriting with concrete syntax is not only feasible, but also convenient. This is due to the seamless integration of context-free syntax definition, parse forests and rewrite rules. Based on a large collection of COBOL programs we have presented performance figures of an interpreter executing a collection of simple disambiguation filters.

References

[1] R. op den Akker, B. Melichar, and J. Tarhio. Attribute Evaluation and Parsing. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 187–214. Springer-Verlag, 1991.

[2] J. Aycock. Why Bison is becoming extinct. *ACM Crossroads*, Xrds-7.5:electronic publication, 2002.

[3] J. Aycock and R.N. Horspool. Directly-executable earley parsing. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 299–243, Genova, Italy, 2001. Springer-Verlag.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and Ch. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Second Intl. Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998.

[5] M.G.J. van den Brand. *Pregmatic: A Generator for Incremental Programming Environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[6] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[7] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[8] M.G.J. van den Brand, P.-E. Moreau, and C. Ringeissen. The ELAN environment: a rewriting logic environment based on ASF+SDF technology. In M.G.J van den Brand and R. Lämmel, editors, *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, volume 65. Electronic Notes in Theoretical Computer Science, 2002.

[9] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.

[10] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[11] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The Syntax Definition Formalism SDF - reference manual*, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989.

[12] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell. A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[13] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[14] M. de Jonge and J. Visser. Grammars as contracts. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *LNCS*, pages 85–99, Erfurt, Germany, 2001. Springer-Verlag.

[15] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

[16] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–110. Oxford University Press, 1992.

[17] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M. G. J. van den Brand and D. Parigot, editors, *Proc. Workshop on Language Descriptions, Tools and Applications*, volume 44 of *Electronic Notes in Theoretical Computer Science*, pages 28–52. Elsevier Science Publishers, 2001.

[18] R. Lämmel and C. Verhoef. Semi-Automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[19] P.J. Landin. The next 700 programming languages. In *CACM*, volume 9, pages 157–165, March 1966.

[20] D.A. Turner. *SASL Language Manual*. University of Kent, Canterbury, 1979.

[21] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[22] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.

[23] E. Visser. Meta-programming with concrete object syntax. In D. Batory and C. Consel, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*. Springer-Verlag, 2002.

[24] T.A. Wagner and S.L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 31–43. ACM Press, 1997.