Type-driven Automatic Quotation of Concrete Object Code in Meta Programs

J.J. Vinju

Centrum voor Wiskunde en Informatica
P.O. Box 94079, NL-1090 GB, Amsterdam, The Netherlands
Jurgen.Vinju@cwi.nl

Abstract. Meta programming can be facilitated by the ability to represent program fragments in concrete syntax instead of abstract syntax. The resulting meta programs are more self-documenting. One caveat in concrete meta programming is the syntactic separation between the meta language and the object language. To solve this problem, many meta programming systems use quoting and anti-quoting to indicate precisely where level switches occur. These "syntactic hedges" can obfuscate the concrete program fragments. This paper describes an algorithm for inferring quotes, such that the meta programmer no longer needs to explicitly indicate transitions between the meta and object languages.

1 Introduction

Programs that manipulate programs as data are called *meta programs*. Examples of meta programs are compilers, source-to-source translators, type-checkers, documentation generators, refactoring tools, and code generators. We call the language that is used to manipulate programs the *meta language*, and the manipulated language the *object language*. Meta programming is the method for implementing automated software engineering tools.

Any general purpose programming language can be used to write meta programs. The object program fragments are represented using the data type constructs available in the meta language. An old idea to facilitate meta programming is the use of *concrete syntax* to represent program fragments [1]. Using concrete syntax, as opposed to abstract syntax, all program fragments in a meta program are represented in the syntax of the object language [2,3]. Concrete syntax combines the readability of the string representation with the structural and type-safe representation of abstract syntax trees. The meta programmer embeds the actual program fragments literally in his meta program, but the underlying representation of these fragments is an abstract syntax tree. The resulting meta programs are more self-documenting because "what you see is what you manipulate".

One caveat in concrete meta programming is the syntactic separation between the meta language and the object language. Conventional scanning and parsing technologies have a hard time distinguishing the two levels. To solve

this problem, many meta programming systems use quoting and anti-quoting to indicate precisely where level switches are made. To further guide the system, the user is sometimes obliged to explicitly mention the type of the following program fragment. These "syntactic hedges" help the parser, but they can obfuscate the concrete program fragments. In practice, it leads to programmers avoiding the use of concrete syntax because the benefit becomes much less clear when it introduces more syntactic clutter than it removes. We would like to *infer* the transitions between the object and meta languages automatically without asking the user to express the obvious.

Contributions and road-map. This paper contributes by removing the technical need for quotes and anti-quotes. We first explore meta programming with concrete syntax in some detail by describing a number of existing systems that implement it (Section 2). We then introduce an algorithm that automatically detects transitions from meta language to object language (Section 3). The architecture around this algorithm is based on scannerless generalized parsing and a separate type-checking phase that both have been described earlier [4, 5]. By making the transitions between meta language and object language invisible we introduce parsing challenges: ambiguous and cyclic grammars. In Section 4 we address these issues. Sections 5 and 6 describe experience and conclusions respectively.

2 The syntax of program fragments in meta programming

Plain Java. Suppose we use Java as a meta programming language to implement a Java code generator. Consider the following method that generates a Java method.

```
String buildSetter(String name, String type) {
  return "public void set" + name + "(" + type + " arg)\n"
       + "  this." + name + " = arg; }\n"
}
```

The string representation is unstructured, untyped and uses quotes and anti-quotes. There is no guarantee that the output of this method is a syntactically correct Java method. However, the code fragment is immediately recognizable as a Java method. The following Java code applies a more structured method to construct the same fragment:

```
String buildSetter(String name, String type) {
  Method method = method(
       publicmodifier(), voidType(), identifier("set" + name),
       arglist(formalarg(classType(type),identifier("arg"))),
       statlist(stat(assignment(
             fieldref(identifier("this"),identifier(name)),
             expression(identifier("arg"))))));
  return method.toString();
}
```

This style uses a number of methods for constructing an abstract syntax tree in a bottom-up fashion. If the used construction methods are strictly typed, this style exploits the Java type system to obtain a syntactically correct result. That means that if all `toString()` methods of the abstract representation are correct, then the new expression will also generate syntactically correct Java code.

The Jakarta Tool Suite. This is the first system we describe that employs concrete syntax. JTS is designed for extending programming languages with domain specific constructs. It implements and extends ideas of intentional programming and work in the field of syntax macros [6].

The parser technology used in JTS is based on a separate lexical analyzer and an LL parser generator. This restricts the number of language extensions that JTS accepts. The program fragments in JTS are quoted with *explicit typing*. For selected non-terminals there is a named quoting and anti-quoting operator, like `mth{...}` and `$id(...)` in the following example:

```
public FieldDecl buildSetter(String name, String type) {
   QualifiedName methodName = new QualifiedName("set" + name);
   QualifiedName fieldName = new QualifiedName(name);
   QualifiedName typeName = new QualifiedName(type);
   return mth{public void $id(methodName) ($id(typeName) arg) {
               this.$id(fieldName) = arg; } } }
```

Concrete syntax in ML. In [7] an approach for adding concrete syntax to ML is described. This system also uses quotation operators. It employs scannerless parsing with Earley's generalized parsing algorithm. Disambiguation of the meta programs with program fragments is obtained by the following:

- Full integration of the parser and type-checker of ML: a context-sensitive parser. All type information can be used to guide the parser. Only type correct derivations are recognized, such that quoting operators do not often need explicit types like in JTS.
- In case the type-checking parser cannot decide, the user may explicitly annotate quoting operators with types, like in JTS.

This system is able to provide typing error messages instead of parse errors. Both the level of automated disambiguation, and the level of the error messages are high. The following example shows how anonymous quoting (`[|...|]`) and anti-quoting (`^...`) are used to indicate transitions:

```
fun buildSetter name type =
            [| public void ^(concat "set" name) (^type arg) {
                 this.^name = arg; } |]
```

Meta-Aspect/J. This is a tool for meta programming Aspect/J programs in Java [8]. It employs context-sensitive parsing, in a manner similar to the approach taken for ML. As a result, this tool also does not need explicit typing.

```
MethodDec buildSetter(String name, String type) {
  String methodName = "set" + name;
  return `[public void #methodName (#type arg) {
              this.#name = arg; } ]; }
```

Note that Meta Aspect/J offers a fixed combination of one meta language (Java) with one single object language (Aspect/J), while the other systems combine one meta language with many object languages.

ASF+SDF. This is a specialized language for meta programming with concrete syntax. The implementation of ASF+SDF is based on scannerless generalized LR parsing (SGLR) [4] and conditional term rewriting [9]. The syntax of the object language is defined in the SDF formalism. Then rewrite rules defined in ASF implement appropriate transformations, using concrete syntax. The SGLR algorithm takes care of a number of technical issues that occur when parsing concrete syntax:

- It accepts all context-free grammars, which are closed under composition. This allows the combination of any meta language with any object language.
- Due to scannerless parsing, there are no implicit global assumptions like longest match of identifiers, or reserved keywords. Such assumptions would influence the parsing of meta programs. The combined language would have the union set of reserved keywords, which is incorrect in both separate languages.
- Unlimited lookahead takes care of local conflicts in the parse table.

The following rephrases the examples of the introduction in ASF+SDF:

```
context-free syntax
   buildSetter(Identifier, Type) -> Method
variables
  "Name"   -> Identifier
  "Type"   -> Type
equations
[] buildSetter(Name , Type ) =
   public void set ++ Name (Type arg) {
     this.Name = arg; }
```

ASF+SDF does not have quoting, or anti-quoting. There are two reasons for this. Firstly, within program fragments no nested ASF+SDF constructs occur that might overlap or interfere. Secondly, the ASF+SDF parser is designed in a very specific manner. It only accepts type correct programs because a specialized parser is generated for each ASF+SDF module. The type system of ASF+SDF requires that all equations are *type preserving*. To enforce this rule, a special production is generated for each user-defined non-terminal `X`: `X "=" X -> Equation`. So instead of having one `Term "=" Term -> Equation` production, ASF+SDF generates specialized productions to parse equations. After this syntax generation, the fixed part of ASF+SDF is added. That part

contains the skeleton grammar in which the generated syntax for `Equation` is embedded.

The ASF+SDF example shown above has some syntactic ambiguity. For example, the meta variable `Type` may be recognized as a Java class name, or as a meta variable. Another ambiguity is due to the following user-defined *injection* production: `Method -> Declaration`. Thus, the equation may range over the `Declaration` type as well as over the `Method` type. To disambiguate, ASF+SDF prefers to recognize declared meta variables over object syntax identifiers, and shorter derivations over longer derivations. We call these two preferences the *meta disambiguation rules* of ASF+SDF. This design offers the concrete syntax functionality we seek, but the assumptions that are made limit its general applicability:

- The type system of the meta language must be expressible as a context-free grammar. Consequently, higher-order functions or parametric polymorphism are not allowed.
- Typing errors are reported as parsing errors which makes developing meta programs difficult.

Stratego. This is a meta programming language based on the notion of rewrite rules and strategies [10]. The concrete object syntax feature of Stratego is also based on SGLR, but the separation between the meta language and the object language is done by quoting and anti-quoting. The programmer first defines quotation and anti-quotation notation syntax herself, and then the object language is combined with the Stratego syntax. After parsing, the parse tree of the meta program is mapped automatically to normal Stratego abstract syntax [11]. This is natural for Stratego, since it has no type system to guide parsing, and nested meta programs are allowed in object fragments.

The following example defines the syntax of quotation operators for some Java non-terminals, with and without explicit types:

```
context-free syntax
         "|[" Method "]|" -> Term        {cons("toMetaExpr")}
  "Method |[" Method "]|" -> Term        {cons("toMetaExpr")}
  "~" Term                -> Identifier  {cons("fromMetaExpr")}
  "~id" Term              -> Identifier  {cons("fromMetaExpr")}
variables
  "type" -> Type
strategies
builderSetter(|name, type) =
    !|[public void ~<conc-strings> ("set", name)(type arg) {
         this.~name = arg; } ]|
```

The productions' "cons" attributes are used to guide the automated mapping to Stratego abstract syntax. The ambiguities that occur in ASF+SDF due to injections also occur in Stratego, but the user can always use the explicitly typed quoting operators. In the example, we used both Stratego syntax, like the `!` operator and the `conc-strings` library strategy, and Java object syntax.

To indicate the difference, we also used an implicit meta variable for the type argument, and an explicitly anti-quoted variable for the field name that we set.

Stratego leaves part of implementing concrete syntax, namely combining the meta language with the object language, to the user. The use of quoting operators makes this job easier, but the resulting meta programs contain many quoting operators. Questions the user must be able to answer are:

- For which non-terminals should quotation operators be defined.
- When should explicit typing be used.
- What quotation syntax would be appropriate for a specific non-terminal.

If not carefully considered, the answers to these questions might differ for different meta programs that manipulate the same object language. The solution to this problem is to let an expert define the quotation symbols for selected object languages, and put these definitions in a library.

TXL. TXL [12] is a meta programming language that uses backtracking to generalize over deterministic parsing algorithms. TXL has a highly structured syntax, which makes extra quoting unnecessary. Every program fragment is enclosed by a certain operator. The keywords of the operators are syntactic hedges for the program fragments:

```
function buildMethod Name [Identifier] Type [Type]
  replace M [Method]
    construct MethodName [Identifier]
       set [+ Name]
  by
    public void MethodName (Type arg) {
       this.Name = arg; }
end function
```

The example shows how code fragments and the first occurrence of fresh variables are explicitly typed. The `[...]` anti-quoting operator is used for explicit typing, but it can also contain other meta level operations, such as recursive application of a rule or function. Keywords like `construct`, `replace`, and `by` cannot be used inside program fragments, unless they are escaped.

Although technically TXL does use syntactic hedging, the user is hardly aware of it due to the carefully designed syntax of the meta language. Compared to other meta programming languages, TXL has more keywords.

Discussion. Table 1 summarizes the concrete meta programming systems just discussed. The list is not exhaustive, there are many more meta programming systems, or language extension systems available. Clearly the use of quoting and anti-quoting is a common design decision for meta programming systems with concrete syntax. Explicit typing is also used in many systems. Type-safety is implemented in most of the systems described. From studying the above systems, we draw the following conclusions:

	ASF	Stratego	ML	JTS	TXL	MAJ
Typed	+	−	+	+	+	+
Implicit quoting	+	Opt.	−	−	+	−
No type annotations	+	Opt.	−	−	−	+
Nested meta code	−	+	+	−	+	−

Table 1. Concrete syntax in several systems.

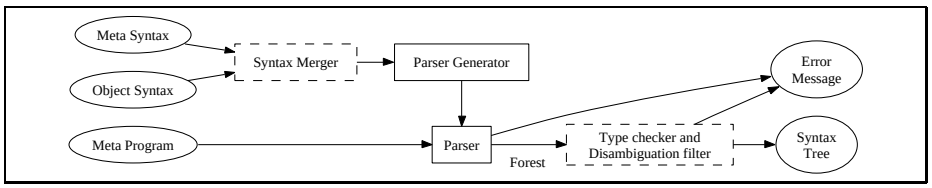

Fig. 1. Overview: parsing concrete syntax using type-checking to disambiguate.

- The more *typing context* provided by the meta programming language, the less explicit quoting operators are necessary.
- It is hard to validate the claim that less quotation and anti-quotation is better in all cases. Possibly, this boils down to a matter of taste. Evidently *unnecessary* syntactic detail harms programmer productivity, but that argument just shifts the discussion to what is necessary and what is not. A hybrid system that employs both quote inferencing and explicit quoting would offer the freedom to let the user choose which is best.
- A shortcoming of many systems that employ concrete syntax is that the error messages that they are able to provide are not very informative.

Our goal is to design a parsing architecture that can recognize code fragments in concrete syntax, without syntactic hedges, embedded in meta programming languages with non-trivial expression languages with strict type systems. As an aside, we note that *syntax highlighting* object code differently and the use of *optional quoting operators* are very practical features, but we consider them to be orthogonal to our contribution.

3 Architecture

We start with a fixed syntax definition for a meta language and a user-defined syntax definition for an object language. In Fig. 1 the general architecture of the process starting from these two definitions and a meta program, and ending with an abstract syntax tree is depicted. The first phase, the syntax merger, combines the syntax of the meta language with the syntax of the object language.

The second phase parses the meta program using SGLR [4]. Generalized parsing algorithms do not complain about ambiguities or cycles. In case of ambiguity they produce a compact representation of a "forest" of trees. This enables this architecture in which the disambiguation process is merged with the type

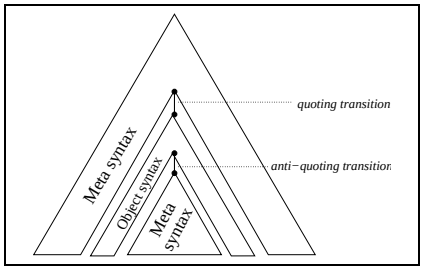

Fig. 2. A parse tree may contain both meta and object productions, where the transitions are marked by quoting and unquoting transition productions.

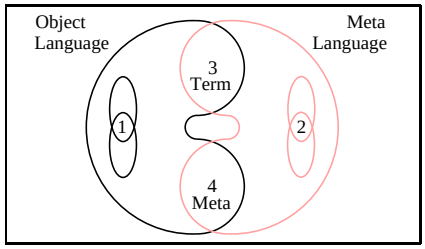

Fig. 3. Classification of ambiguities after joining a meta language with an object language.

checking algorithm of the meta language rather than integrated in its parsing algorithm.

The final phase type-checks and disambiguates the parse forest, filtering out type-incorrect trees. This architecture is consistent with the idea of disambiguation by filtering as described by [13], which has been applied earlier [4, 5, 14].

3.1 Syntax transitions

The syntax merger creates a new syntax module, importing both the meta syntax and the object syntax. We assume there is no overlap in non-terminals between the meta syntax and the object syntax, or that renaming is applied to accomplish this. It then adds productions that link the two layers automatically. For every non-terminal X in the object syntax the following productions are generated: X `-> Term` and `Term ->` X`,` where `Term` is a unique non-terminal selected from the meta language. For example, for Java, the `Term` non-terminal would be `Expression`, because expressions are the way to build data structures in Java.

We call these productions the *transitions* between meta syntax and object syntax. They replace any explicit quoting and unquoting operators. For clarity we will call the transitions to meta syntax the *quoting transitions* and the transitions to object syntax the *anti-quoting transitions*. Figure 2 illustrates the intended purpose of the transitions: nesting object language fragments in meta programs, and nesting meta language fragments again in object language fragments.

The collection of generated transitions from and to the meta language are hazardous. They introduce many ambiguities, including cyclic derivations. An *ambiguity* arises when more than one derivation exists for the same substring with the same non-terminal. Intuitively, this means there are several interpretations possible for the same substring. A *cycle* occurs in derivations if and only if a non-terminal can produce itself without consuming terminal symbols. To get a correct parser for concrete meta programs without quoting, we must resolve all ambiguities introduced by the transitions between meta and object syntax.

Figure 3 roughly classifies the ambiguities that may occur. Note that ambiguities from different classes may be nested.

Class 1: Ambiguity in the object language itself. This is an artifact of the user-defined syntax of the object language. Such ambiguity must be ignored, since it is not introduced by the syntax merger. The C language would be a good example of an ambiguous object language, with its overloaded use of the `*` operator for multiplication and pointer dereference.

Class 2: Ambiguity of the meta language itself. This is to be ignored too, since it is not introduced by the syntax merger. Usually, the designer of the meta language will have to solve such an issue separately.

Class 3: Ambiguity directly via syntax transitions. The `Term` non-terminal accepts all sub languages of the object language. Parts of the object language that are nicely separated in the object grammar, are now overlaid on top of each other. For example, the *isolated* Java code fragment `i = 1` could be a number of things including an assignment statement, or the initializer part of a declaration.

Class 4: Object language and meta language overlap. Certain constructs in the meta language may look like constructs in the object language. In the presence of the syntax transitions, it may happen that meta code can also be parsed as object code. For example, this hypothetical Java program constructs some Java declarations: `Declarations decls = int a; int b;`. The `int b;` part can be in the meta program, or in the object program.

4 Disambiguation filters

We will explicitly ignore ambiguity classes 1 and 2, such that the proposed disambiguation filters do not interfere with the separate definitions of the meta language and the object language. We will analyze ambiguity classes 3 and 4, and explain how a disambiguating type-checker will either resolve these ambiguities.

4.1 Disambiguation by type-checking

Type-checking is a phase in compilers where it is checked if all operators are applied to compatible operands. Traditionally, a separate type-checking phase takes an abstract syntax tree as input and one or more symbol tables that define the types of all declared and built-in operators. The output is either an error message, or a new abstract syntax tree that is decorated with typing information [15]. Other approaches incorporate type-checking in the parsing phase [7, 16] to help the parser avoid conflicts. We do the exact opposite, the parser is kept simple while the type-checker is extended with the ability to deal with alternative parse trees [5].

Figure 4 shows the internal organization of a disambiguating type-checker. Type-checking forests to filter them is a natural extension of normal type-

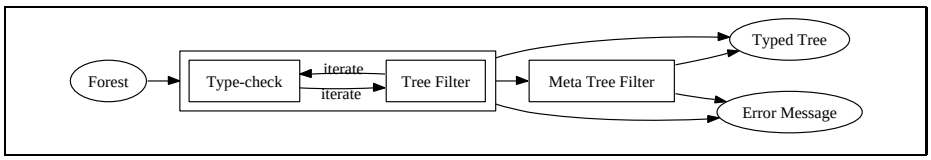

Fig. 4. The organization of the type-checking and disambiguation approach.

checking of trees. A forest may have several sub-trees that correspond to different interpretations of the same input program. Type-checking a forest is the process of selecting the single type correct tree. If no single type correct tree is available then we deal with the following two cases:

- No type correct abstract syntax tree is available; present the collection of error messages corresponding to all alternative trees,
- Multiple type correct trees are available; present an error message explaining the alternatives.

Note that resolving the ambiguities caused by syntax transitions resembles type-inference for polymorphic functions [17]. The syntax transitions can be viewed as overloaded (ad-hoc polymorphic) functions. There is one difference: the forest representation already provides the type-inference algorithm with the set of instantiations that is locally available, instead of providing one single abstract tree that has to be instantiated.

Regarding the feasibility of this architecture, recall that the amount of nodes in a GLR parse forest can be bounded by a polynomial in the length of the input string [18, 19]. This is an artifact of smart sharing techniques for parse forests produced by generalized parsers. Maximal sub-term sharing [20] helps to lower the average amount of nodes even more by sharing all duplicated sub-derivations that are distributed across single and multiple derivations in a parse forest. However, the scalability of this architecture still depends on the size of the parse forest, and in particular the way it is traversed. A maximally shared forest may still be traversed in an exponential fashion. Care must be taken to prevent visiting unique nodes several times. We use *memoization* to make sure that each node in a forest is visited only once.

4.2 Class 3. Ambiguity directly via syntax transitions

We further specialize this class intro four parts:

Class 3.1: Cyclic derivations. These are derivations that do not produce any terminals and exercise syntax transitions both to and from the meta grammar. For example, every X has a direct cycle by applying X `-> Term` and `Term ->` X`.`

Class 3.2: Meaningless coercions. These are derivations that exercise the transition productions to cast any X from the object language into another Y. Namely, every X can be produced by any other Y now by applying `Term ->` X and Y `-> Term.`

Class 3.3: Ambiguous quoting transitions. Several X `->` `Term` are possible from different Xs. The ambiguity is on the `Term` non-terminal. For any two non-terminals X and Y that produce languages with a non-empty intersection, the two productions X `->` `Term` and Y `->` `Term` can be ambiguous.

Class 3.4: Ambiguous anti-quoting transitions. Several `Term` `->` X are possible, each to a different X. For any two productions of the object language that produce the same non-terminal this may happen. A `->` X and B `->` X together introduce an anti-quoting ambiguity with a choice between `Term` `->` A and `Term` `->` B.

In fact, classes 3.1 and 3.2 consist of degenerate cases of ambiguities that would also exist in classes 3.2 and 3.3. We consider them as a special case because they are easier to recognize, and therefore may be filtered with less overhead. The above four subclasses cover all ambiguities caused directly by the transition productions. The first two classes require no type analysis, while the last two classes will be filtered by type checking.

Class 3.1. Dealing with cyclic derivations The syntax transitions lead to cycles in several ways. The most direct cycles are the immediate application of an anti-quoting transition after a quoting transition. Any cycle, if introduced by the syntax merger, always exercises at least one production X `->` `Term`, and one production `Term` `->` Y for any X or Y [21].

Solution 1. The first solution is to filter out cyclic derivations from the parse forest. With the well known `Term` non-terminal as a parameter we can easily identify the newly introduced cycles in the parse trees that exercise cyclic applications of the transition productions. A single bottom-up traversal of the parse forest that detects cycles by marking visited paths is enough to accomplish this. With the useless cyclic derivations removed, what remains are the useful derivations containing transitions to and from the meta level.

We have prototyped solution 1 by extending the ASF+SDF parser with a cycle filter. Applying the prototype on existing specifications shows that for ASF+SDF such an approach is feasible. However, the large amount of meaningless derivations that are removed later do slow down the average parse time of an ASF+SDF module significantly. To quantify, for smaller grammars with ten to twenty non-terminals we witnessed a factor of 5, while for larger grammars with much more non-terminals we witnessed factors of 20 times slow down.

Solution 2. Instead of filtering the cycles from the parse forest, we can prevent them by filtering reductions from the parse table. This technique is based on the use of a disambiguation construct that is described in [4]. We use *priorities* to remove unwanted derivations, in particular we remove the reductions that complete cycles. The details of this application of priorities to prevent cycles are described in a technical report [21]. The key is to automatically add the following priority for every object grammar non-terminal X: X `->` `Term` `>` `Term` `->` X`.`

Because priorities are used to remove reductions from the parse table the cyclic derivations do not occur at all at parsing time.

Prototyping the second scheme resulted in a considerable improvement of the parsing time. The parsing time goes back to almost the original performance. However parse table generation time slows down significantly. So when using solution 2, we trade some compilation time efficiency for run time efficiency. In a setting with frequent updates to the object grammar, it may pay off to stay with solution 1.

Class 3.2. Dealing with meaningless coercions For every pair of non-terminals X and Y of the object language that produce languages that have a non-empty intersection, an ambiguity can be constructed by applying the productions `Term -> X` and Y `-> Term.` Effectively, such a derivation casts an Y to an X, which is a meaningless coercion.

These ambiguities are very similar to the cyclic derivations. They are meaningless derivations occurring as a side-effect of the introduction of the transitions. Every direct nesting of an unquoting and a quoting transition falls into this category. As such they are identifiable by the structure of the tree, and a simple bottom-up traversal of a parse forest is able to detect and remove them. No type information is necessary for this. As an optimization, we can again introduce *priorities* to remove these derivations earlier in the parsing architecture.

Class 3.3. Dealing with ambiguous quoting So far, no type checking was needed to filter the ambiguities. This class however is more interesting. The X `-> Term` productions allow everything in the object syntax to be `Term`. If there are any two non-terminals of the object language that generate languages with a non-empty intersection, and a certain substring fits into this intersection we will have an ambiguity. This happens for example with all injection productions: X `->` Y`,` since the language produced by X is the same as the language produced by Y.

An ambiguity in this class consists of the choice of nesting an X, or an Y object fragment into the meta program. So, either by X `-> Term` or by Y `-> Term` we transit from the object grammar into the meta grammar. The immediate typing context is provided by the surrounding meta code. Now suppose this context enforces an X. Disambiguation is obtained by removing all trees that do not have the X `-> Term` production at the top.

The example in Fig. 5 is a forest with an ambiguity caused by the injection problem. Suppose that from a symbol table it is known that `f` is declared to be a function from `Expression` to `Identifier`. This provides a type-context that selects the transition to `Expression` rather than the transition to `Identifier`.

Class 3.4. Dealing with ambiguous anti-quoting This is the dual of the previous class. The `Term -> X` productions cause that at any part of the object language can contain a piece of meta language. We transit from the meta

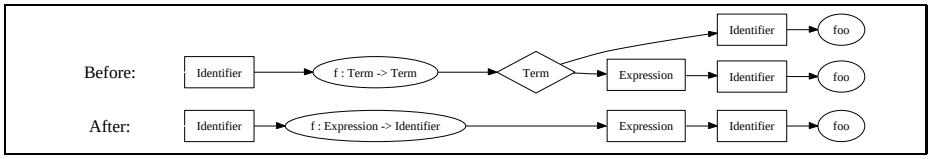

Fig. 5. An abstract syntax forest (drawn from left to right) is disambiguated by using a type declaration for the function `f`.

grammar into the object grammar. The only pieces of meta language allowed are produced by the `Term` non-terminal. The typing context is again provided by the meta language, but now from below. Suppose the result type of the nested meta language construct is declared X, then we filter all alternatives that do not use the `Term -> X` transition.

Discussion To implement the above four filters a recursive traversal of the forest is needed. It applies context information on the way down and brings back type information on the way back. Note that the typing contexts necessary for the above may be inconclusive. For example, with *overloaded* methods in Java a type-checker may not be able to decide which is which. In general, when several ambiguities remain, a disambiguation type-checker may either choose to precisely report the conflict, or continue and use some of the filters discussed in the following.

4.3 Class 4. Object language and meta language overlap

The most common example in this class is how to separate meta variables from normal identifiers in the object syntax (Section 2). Other examples are more complex: the meta and object program fragments must accidentally have exactly the same syntax, and both provide type-correct interpretations. The following example illustrates this. The meta language is ASF+SDF, and the object language is Java:

```
equations
[] int[] foo = int[]
[] foo = bar
```

```
equations
[] int[] foo = int[][] foo = bar
```

The overlapping language constructs are the fragments: "`[]`" and "`=`". For ASF+SDF, the "`[]`" is an empty equation tag and "`=`" is the equation operator, while in Java "`[]`" is a part of an array declarator and "`=`" is the initializer part. The parser returns two alternative interpretations. The first has two equations, the second only one. By using suggestive layout, and printing the ASF+SDF symbols in *italics*, we illustrate how the right-hand side of the first rule can be extended to be a two dimensional array declarator that is initialized by `bar`: the combination of `"[]"` overlapping with array declarators and `"="` overlapping with variable initializers leads to the ambiguity. Both interpretations are

syntactically correct and type-correct. Note that the above example depends on a particular Java object grammar.

To solve this class of rare and hard to predict ambiguities we introduce a separate meta disambiguation phase (Fig. 4). A number of implicit but obvious rules will provide a full separation between meta and object language. The rules are not universal. Each meta programming language may select different ones applied in different orders. Still, the design space can be limited to a number of choices:

Rule 1: Prefer/avoid declared meta identifiers over object identifiers.
Rule 2: Prefer simpler/more complex object language derivations.
Rule 3: Maximize/minimize the number of meta language productions.
Rule 4: Propose explicit quoting to the user.

Rule 1 is a generalization of the variable preference rule (Section 2). All meta level identifiers, such as function names and variables are preferred. This rule may involve counting the number of declared meta identifiers in two alternatives and choosing the alternative with the least or most meta identifiers.

Rule 2 is needed only when the type context is not specific down to a first order type, but does impose some constraints. This can happen in systems with polymorphic operators, like Haskell functions, or the overloaded methods of Java, or the equations of ASF+SDF. For example: a function with type `f: a -> a -> b` maps two objects of any type `a` to another object of type `b`. Even though the function is parameterized by type, the first two arguments must be of the same type. In the presence of injections like X `->` Y`,` any X can also be a Y. If the first argument of the function `f` can be parsed as an X, it could also be parsed as a Y. Rule 2 then decides whether the "simpler" X or the more "complex" Y interpretation is picked.

Rule 3 expresses that object language fragments should be either as short, or as long as possible. The more meta productions are used, the shorter object fragments become. This takes care of our earlier example involving the `"[]"` and `"="`.

Rule 4 If Rule 3 fails, Rule 4 provides the final fail-safe to all ambiguities introduced by merging the meta and object syntax.

Discussion The above rules have a heuristic nature and their semantics depend on the order in which they are applied. They should always be applied after the type-checker, such that only degenerate cases are subject to these rules. Another viewpoint is to avoid the application of these heuristics altogether, and propose explicit disambiguations to the user when necessary.

5 Experience

Parsing. This work has been applied to parsing ASF+SDF specifications. First the syntax of ASF+SDF was extended to make the meta language more complex: a generic expression language was added that can be arbitrarily nested with

object language syntax. Before there were only meta variables in ASF+SDF, now we can nest meta language function calls at arbitrary locations into object language patterns, even with parametric polymorphism. Then a syntax merger was developed that generates the transitions, and priorities for filtering cycles. The generated parsers perform efficiently, while producing large parse forests.

Type based disambiguation. Both experience with post-parsing disambiguation filters in ASF+SDF [14], and the efficient implementation of type-inference algorithms for languages as Haskell and ML suggests that our cycle removal and type-checking disambiguation phase can be implemented efficiently. Knowing the polynomial upper-bound for the size of parse forests, we have implemented a type-checker for ASF+SDF that applies the above described basic disambiguation rules. Note that the equation operator of ASF+SDF can be viewed as a parametric polymorphic operator, but we have not yet experimented with user-defined polymorphic parameters.

Furthermore, in [5] we described a related disambiguation architecture that also employs disambiguation by type checking. In this approach we show how such an architecture can remove the need for explicitly typed quotes, but not the need for explicit quoting in general. We applied the disambiguation by type checking design pattern to Java as a meta programming language. A Java type checker is used to disambiguate object language patterns. As reported, such an algorithm performs very well. We further improve on these results in this paper, removing the need for explicit quotes altogether.

6 Conclusion

An architecture for parsing meta programs with concrete syntax was presented. By using implicit transitions syntactic hedges are avoided. We offered a "quote inference" algorithm, that allows a programmer to remove syntactic hedges if so desired. Inferring quotes is a feasible algorithm due to sharing and memoization techniques of parse forests. We use a three-tier architecture, that efficiently detects syntax transitions and then resolves ambiguities, including cycles, or returns precise error messages. Resolving ambiguities occurs either on simple structural arguments, or on typing arguments, or optionally based on heuristics.

References

1. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I.: LISP 1.5 Programmer's Manual. The MIT Press, Cambridge, Mass. (1966)
2. Sellink, M., Verhoef, C.: Native patterns. In Blaha, M., Quilici, A., Verhoef, C., eds.: Proceedings of the Fifth Working Conference on Reverse Engineering, IEEE Computer Society Press (1998) 89–103
3. Heering, J., Hendriks, P., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN Notices **24** (1989) 43–75

4. Brand, M.v.d., Scheerder, J., Vinju, J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In Horspool, R.N., ed.: Compiler Construction (CC'02). Volume 2304 of LNCS., Springer-Verlag (2002) 143–158
5. Bravenboer, M., Vermaas, R., Vinju, J., Visser, E.: Generalized type-based disambiguation of meta programs with concrete object syntax. In: Generative Programming and Component Engineering (GPCE). (2005) to appear.
6. Leavenworth, B.M.: Syntax macros and extended translation. Commun. ACM **9** (1966) 790–793
7. Aasa, A., Petersson, K., Synek, D.: Concrete syntax for data objects in functional languages. In: Proceedings of the 1988 ACM conference on LISP and functional programming, ACM Press (1988) 96–105
8. Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ programs with Meta-AspectJ. In Karsai, G., Visser, E., eds.: Generative Programming and Component Engineering: Third International Conference, GPCE 2004. Volume 3286 of Lecture Notes in Computer Science., Vancouver, Canada, Springer (2004) 1–19
9. Brand, M., Deursen, A., Heering, J., Jong, H., Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In Wilhelm, R., ed.: CC'01. Volume 2027 of LNCS., Springer-Verlag (2001) 365–370
10. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In Middeldorp, A., ed.: RTA'01. Volume 2051 of LNCS., Springer-Verlag (2001) 357–361
11. Visser, E.: Meta-programming with concrete object syntax. In Batory, D., Consel, C., eds.: Generative Programming and Component Engineering (GPCE'02). Volume 2487 of LNCS., Springer-Verlag (2002)
12. Cordy, J., Halpern-Hamu, C., Promislow, E.: TXL: A rapid prototyping system for programming language dialects. Computer Languages **16** (1991) 97–107
13. Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. In Pighizzini, G., San Pietro, P., eds.: Proc. ASMICS Workshop on Parsing Theory, Milano, Italy, Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano (1994) 1–20
14. Brand, M., Klusener, S., Moonen, L., Vinju, J.: Generalized Parsing and Term Rewriting - Semantics Directed Disambiguation. In Bryant, B., Saraiva, J., eds.: Third Workshop on Language Descriptions Tools and Applications. Volume 82 of Electronic Notes in Theoretical Computer Science., Elsevier (2003)
15. Aho, A., Sethi, R., Ullman, J.: Compilers. Principles, Techniques and Tools. Addison-Wesley (1986)
16. Paakki, J.: Attribute grammar paradigms: a high-level methodology in language implementation. ACM Comput. Surv. **27** (1995) 196–255
17. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
18. Johnson, M.: The computational complexity of GLR parsing. In Tomita, M., ed.: Generalized LR Parsing. Kluwer, Boston (1991) 35–42
19. Billot, S., Lang, B.: The structure of shared forests in ambiguous parsing. In: Proceedings of the 27th annual meeting on Association for Computational Linguistics, Morristown, NJ, USA, Association for Computational Linguistics (1989) 143–151
20. Brand, M., Jong, H., Klint, P., Olivier, P.: Efficient Annotated Terms. Software, Practice & Experience **30** (2000) 259–291
21. Vinju, J.: A type driven approach to concrete meta programming. Technical Report SEN-E0507, CWI (2005)