




Centrum voor Wiskunde en Informatica

View metadata, citation and similar papers at core.ac.uk

brought to you by  CORE

provided by CWI's Instituut

REPORTRAPPORT

SEN

Software Engineering



Software ENgineering

A type-driven approach to concrete meta programming

J.J. Vinju

REPORT SEN-E0507 APRIL 2005

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).

CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2005, Stichting Centrum voor Wiskunde en Informatica

P.O. Box 94079, 1090 GB Amsterdam (NL)

Kruislaan 413, 1098 SJ Amsterdam (NL)

Telephone +31 20 592 9333

Telefax +31 20 592 4199

ISSN 1386-369X

A type-driven approach to concrete meta programming

ABSTRACT

Applications that manipulate programs as data are called meta programs. Examples of meta programs are compilers, source-to-source translators and code generators. Meta programming can be supported by the ability to represent program fragments in concrete syntax instead of abstract syntax. The resulting meta programs are far 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 manipulated language. Conventional scanning and parsing technologies have a hard time distinguishing the two levels. To aid in solving this problem, many meta programming systems use quoting and escaping to indicate precisely where level switches occur. These "syntactic hedges" can heavily obfuscate the concrete program fragments. In practice, it leads to programmers avoiding the use of concrete syntax for the bigger or more deeply nested program fragments. This paper contributes by removing the technical need for quotes and escapes, allowing more "what you see is what you manipulate" than before in meta programming applications. The method is based on typing mechanisms.

2000 Mathematics Subject Classification: n/a

1998 ACM Computing Classification System: D3.3, D1.1

Keywords and Phrases: Concrete syntax, parsing, type checking

A Type-driven Approach to Concrete Meta Programming

Jurgen J. Vinju

Centrum voor Wiskunde en Informatica (CWI),
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands,
Jurgen.Vinju@cwi.nl

Abstract. Applications that manipulate programs as data are called *meta programs*. Examples of meta programs are compilers, source-to-source translators and code generators. Meta programming can be supported by the ability to represent program fragments in concrete syntax instead of abstract syntax. The resulting meta programs are far 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 manipulated language. Conventional scanning and parsing technologies have a hard time distinguishing the two levels. To aid in solving this problem, many meta programming systems use quoting and escaping to indicate precisely where level switches occur.

These “syntactic hedges” can heavily obfuscate the concrete program fragments. In practice, it leads to programmers avoiding the use of concrete syntax for the bigger or more deeply nested program fragments. This paper contributes by removing the technical need for quotes and escapes, allowing more “what you see is what you manipulate” than before in meta programming applications.

1 Introduction

Applications 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*.

Any general purpose programming language can be used to write meta programs. The program is represented using the data-type constructs present in that language. In general, the data-type used is an abstract syntax tree, but sometimes a string representation seems to be sufficient. Many code generators just print out strings for example, while compilers and type-checkers depend on a more structured representation of a program. Although the string representation is very readable, it is too weak to be a device for modeling program structure. Consequently strings do not scale for use in more complex meta programs; there is not enough compile-time checking.

The main disadvantage of using a general purpose programming language for meta programming is that the original program is encoded in a very complex data-type. The

abstract syntax tree data-type has to be designed in advance, and then the basic functionality to manipulate this data also needs to be developed before the actual manipulation can be expressed.

To relieve the meta programmer from designing abstract syntax trees and the basic operations on them there exist code generators, libraries and domain specific languages for meta programming. These tools can help you to apply the following operations to abstract syntax trees on a higher level of abstraction:

- construction
- deconstruction or matching
- search or traversal
- transformation
- decoration

Some tools provide a typing mechanism to model program structure even more faithfully and strictly. The high level of abstraction offered by these tools brings a meta program conceptually closer to the object program and the intended manipulation. The meta program becomes more self-documenting.

An old idea to facilitate meta programming is the use of *concrete syntax* to represent program fragments [11]. Using concrete syntax, as opposed to abstract syntax, all program fragments in a meta program are represented in the syntax of the object language [14, 8].

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 far more self-documenting because “what you see is what you manipulate”.

Anybody who can program in the object language can understand and even write the embedded program fragments in a concrete meta program. Compare this to learning a complex abstract syntax representation that may be application specific.

Concrete syntax is an orthogonal feature for meta programming tools. For example, it can serve as syntactic sugar on top of a domain specific language that offers the aforementioned meta programming features.

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 escaping to indicate precisely where level switches are made. To further guide the system, sometimes it the user is obliged to explicitly mention the type of the following program fragment.

These “syntactic hedges” help the parser, but they can heavily obfuscate the concrete program fragments. In practice, it leads to programmers avoiding the use of concrete syntax for larger or more deeply nested program fragments. This is regrettable, since the benefit of concrete syntax as opposed to abstract syntax becomes clearer as applications become more complex.

Road-map. This paper contributes by removing the technical need for quotes and escapes, allowing more “what you see is what you manipulate” than before in meta programming applications. We introduce an architecture that automatically detects implicit transitions from meta language to object language (Section 3). This architecture is based on generalized parsing and a separate type-checking phase. By making the transitions

Data-structure	Linear (strings)	vs.	Structured (trees)
Typing	Untyped (homogeneous)	vs.	Typed (heterogeneous)
Syntax	Abstract syntax	vs.	Concrete syntax
Quoting	Explicit quoting	vs.	Implicit quoting
Annotations	Explicit type annotations	vs.	No explicit type annotations
Nested meta code	Not allowed	vs.	Allowed

Table 1. Solution space of code fragment representation in meta programming.

between meta language and object language implicit we introduce two challenges for the parsing technology:

Challenge 1: How to deal with cyclic derivations? (Section 4)

Challenge 2: How to deal with ambiguous derivations? (Section 5)

Sections 6 and 7 describe experience and conclusions respectively. For the sake of reference, we first explore meta programming with concrete syntax in some detail and then describe a number of existing systems that implement it (Section 2).

Exploring the solution space. Table 1 is an overview of the syntactic features of meta programming languages for representing code fragments. Systems that represent program fragments using all the features in the right column are preferable, since there is no syntactic distance between the program fragments and the actual object language.

Suppose we use Java as a meta programming language to implement a Java code generator. Consider the following method that generates a Java method. It is written in four different styles: using strings, some abstract syntax, concrete syntax notation with quotation, and finally concrete syntax without quotation. To demonstrate the expressivity of concrete syntax in a familiar setting, we use an imaginary concrete syntax feature for Java in the last two examples.

String representation.

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

The string representation is unstructured, untyped and uses quotes and escapes. However, the code fragment is immediately recognizable as a Java method.

Abstract syntax representation.

```
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 safety. If the methods use some generic tree data-type, we call it an untyped representation. Notice that the complex Java expression that builds the method is hard to read.

Concrete syntax representation with quoting.

```
String buildSetter(String name, String type) {
    Method method = [[public void "set" + name ( 'type' arg) {
                        this.'name' = arg;
                    }]];
    return method.toString();
}
```

To guide the parser the user explicitly indicates the transition from meta language to object language by quoting (`[[. . .]]`), and the transition from object language to meta language by anti-quoting (`' . . .'`). This style with quoting is close to the string representation in terms of readability. It is structured and typed because the implementation of this syntax would introduce compile-time parsing and type-checking of the generated code fragment with respect to the Java grammar.

Concrete syntax representation without quoting.

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

The final example shows our ideal notation. There is a seamless integration between meta language and object language. This approach automates the quoting and anti-quoting for the user. Compare this to type-inference in functional programming languages; if a type can be inferred automatically, we do not ask the user to type it in. Similarly, in concrete syntax without quotation we want to infer the quotes automatically without asking the user to express the obvious.

2 Concrete meta programming systems

The ASF+SDF system is based on scannerless generalized LR parsing (SGLR) [15, 6] and conditional term rewriting. The syntax of the object language is defined in the SDF formalism, after which rewrite rules in ASF in concrete syntax define appropriate transformations [3]. 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. For example, the combined language would have the union set of reserved keywords, which is incorrect in either separate language.
- Unlimited lookahead takes care of local conflicts in the parse-table.

ASF+SDF does not have quoting, anti-quoting or escaping. 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.

This works by exploiting the one-to-one correspondence between the type system of ASF+SDF, and context-free grammars:

- A type maps to a non-terminal.
- A term constructor (“function”) maps to a production.
- Equations must be type-preserving; this fact is mapped to a set of productions, one for each type, e.g.:

```
Tag Expression "=" Expression -> Equation
Tag Identifier "=" Identifier -> Equation
```

After this mapping, the fixed part of ASF+SDF is added. That part contains, among others, the productions that define equation tags and collections of equations:

```
"[" [A-Za-z0-9]* "]" -> Tag
"equations" Equation* -> Section
```

The following are some ASF+SDF equations that rewrite Java expressions:

```
equations
[] Expr + 0 = Expr
[] this + Str = toString() + Str
```

To parse the above fragment ASF+SDF uses two *meta disambiguation* rules:

Rule 1: Prefer to recognize declared meta variables above object syntax identifiers.

Rule 2: Prefer shorter injection chains.

Meta variables are the only ASF+SDF constructs found inside of program fragments. Rule 1 separates variables from program fragments. For example, `Str` is a declared meta variable, but it could also be recognized as a Java `Identifier` directly. By Rule 1, it is a meta variable if it is declared in ASF+SDF like this:

```
variables
  "Str" -> StringConstant
```

Rule 2 deals with injections (chain rules) in the object syntax. For example, the following rewrite rule is ambiguous:

```
[] Hello = Bye
```

By the injection `Identifier -> Expression` in the Java grammar it rewrites either `Hello` identifiers or `Hello` expressions. Rule 2 makes ASF+SDF choose the former over the latter.

Note that such ambiguities also occur for productions that can simulate injections. For example in `A X B -> Y`, where both `A`, and `B` are non-terminals that optionally produce the empty string. We call this a *quasi-injection* from `X` to `Y`.

Although the above design offers the concrete syntax functionality we seek, the assumptions that are made limit its general applicability:

- The type system of the meta language must be expressible as a context-free grammar. Higher-order functions or parametric polymorphism are not allowed.
- Usually, meta programming languages offer more meta level constructs than meta variables. Consider for example `let` or `case` expressions.
- Typing errors are reported as parsing errors which makes debugging meta programs harder.

In *Stratego* [16] the concrete object syntax feature also uses SDF for syntax definition and SGLR to parse Stratego programs. 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. 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 [17].

By letting the user define the quotation operators, Stratego offers a very explicit way of combining meta language and object language. This is natural for Stratego, since:

- There is no type system, so parsing can not be guided by a type context.
- There are meta operators that could appear nested in a program fragment.

The following are example user-defined quotation and anti-quotation operators for a non-terminal in Java:

```
"[" Expression "]" -> Term      {cons("toTerm")}
"~" Term           -> Expression {cons("fromTerm")}
```

The productions' attributes are used to guide the automated mapping to Stratego abstract syntax.

The ambiguities that occur in ASF+SDF due to injections and quasi-injections also occur in Stratego, but the user can always define a syntactically different quotation operator for every non-terminal. For example, the following user-defined quotation operators explicitly mention the non-terminal of interest to make the difference:

```
"Exp" "[" Expression "]" -> Term {cons("toTerm")}
"Id"  "[" Identifier "]" -> Term {cons("toTerm")}
```

An example code fragment in Stratego is:

```
[simplify] Exp[[ ~Expr + 0 ]] => Expr
[tostring] Exp[[ this + ~Str ]] => Exp [[toString() + ~Str]]
```

The above design leaves a part of implementing concrete syntax, namely combining the meta language with the object language largely to the user. The use of quotation 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 to use explicit typing.
- What quotation syntax will be appropriate for a specific non-terminal.

If not carefully considered, the answers to these questions might differ for different applications for the same object language. A solution proposed in [17] is to generate quoting operators automatically from the syntax definition of the object language.

Concrete syntax in Lazy ML. In [1] an approach for adding concrete syntax to Lazy ML is described. This system also uses quotation operators. It employs scannerless parsing with Earley's algorithm, which is roughly equivalent to SGLR. Disambiguation of the meta programs with program fragments is obtained by:

- Full integration of the parser and type-checker of Lazy ML. All type information can be used to guide the parser. So, only type correct derivations are recognized.
- Overlapping meta variables and the injection problem are partially solved by optionally letting the user annotate meta variables explicitly with their type inside program fragments.

The Jakarta Tool Suite is for extending programming languages with domain specific constructs. It implements and extends ideas of intentional programming, and work in the field of syntax macros [10]. Language extension can be viewed as a specific form of meta programming, with a number of additional features.

The parser technology used in JTS is based on a separate lexical analyzer and a Java port of Yacc. This restricts the number of language extensions that JTS accepts, as opposed to scannerless generalized parsing algorithms.

The program fragments in JTS are quoted with explicit typing. For every non-terminal in there is a named quoting and an unquoting operator, for example:

```
Statement cond = stm{ if(i > 42) $stm(stat) }stm;
```

TXL [7] is also meta programming language. It uses backtracking to generalize over Yacc-like parsing algorithms. TXL has a highly structured syntax, which makes extra quoting not necessary. Every program fragment is enclosed by a certain operator. The keywords of the operators are syntactic hedges for the program fragments:

```
rule collapse
  replace [expression] N [number] + N
  by N * 2
end rule
```

The example shows how code fragments are explicitly typed, and also the first occurrence of fresh variables (N). The [. . .] anti-quoting operator is used for explicit typing, but it can also contain other meta level operations, such as recursive application of

	ASF+SDF	Stratego	Lazy ML	JTS	TXL
Typed	+	-	+	+	+
Implicit quoting	+	-	-	-	-
No type annotations	+	-	-	-	-
Nested meta code	-	+	+	-	+

Table 2. Concrete syntax in several systems.

a rule. The keywords of TXL can be used inside of program fragments using escaping, e.g. `'replace`.

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

Concrete syntax without hedges. Table 2 summarizes the concrete meta programming systems just discussed. The list is not exhaustive, there are many more meta programming systems, or language extension systems out there. Clearly the use of quoting and anti-quoting is a common design decision for meta programming systems. Explicit typing is also used in many systems. Only ASF+SDF does not use quoting or explicit typing. Type-safety is implemented in most of the systems described.

Our goal is to design a parser that can recognize concrete syntax without hedges, embedded in languages with non-trivial expression languages and type-systems.

3 An architecture for parsing concrete meta programs

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 meta and the object language. The second phase parses the meta program. The final phase type-checks and disambiguates the program.

We assume SGLR or an equivalent parsing algorithm. This allows an architecture in which the disambiguation process is merged with the type checker instead of including type-checking in a traditional complicated way inside of the parsing algorithm. The parser returns a “forest” of trees, efficiently representing all possible derivations. After this the type-checker filters out a single type-correct tree or returns a type error. This architecture is consistent with the idea of disambiguation by filtering as described by [9].

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. It then adds productions that link the two layers automatically. For every non-terminal X in the object syntax the following productions are generated (Fig. 2):

$X \rightarrow \text{Tree}$ and $\text{Tree} \rightarrow X$.

We call these productions the *transitions* between meta syntax and object syntax. They replace any explicit quoting operators. The `Tree` non-terminal is an arbitrary

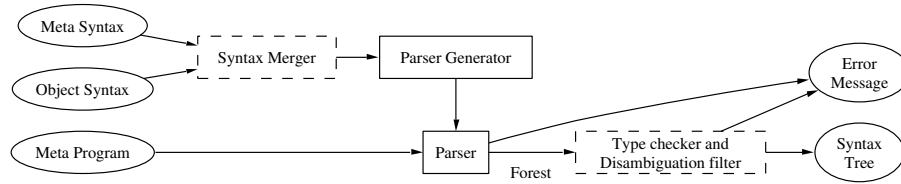


Fig. 1. Overview: parsing concrete syntax using type-checking to disambiguate.

but fixed non-terminal in the meta language that indicates where the concrete syntax fragments are expected. For example, in ASF+SDF the `Tree` non-terminal would be used like this in the definition of the meta language:

```
Tag Tree "=" Tree -> Equation
```

The above collection of generated transitions are hazardous for two reasons. They introduce cyclic derivations (Challenge 1), and ambiguous derivations (Challenge 2). *Cycles* occur in derivations for context-free grammars if and only if a non-terminal can produce itself without consuming terminal symbols. The syntax transitions lead to cycles in several ways, for example:

```
Identifier -> Tree, Tree -> Identifier
Tree -> Identifier, Identifier -> Expression, Expression -> Tree
```

Generalized parsing algorithms do not complain about such cycles, they can produce parse *graphs* when a cycle is present. Still most parser implementations treat cyclic derivations as errors, because many back-ends usually deal with trees, not graphs.

Ambiguities occur when several derivations are produced by the same non-terminal and for the same input string. Generalized parsing algorithms return several sub-trees for an ambiguous sub-string, resulting in a *forest*.

To get a working parser for concrete meta programs without quoting, we must resolve all cycles and ambiguities introduced by the syntax transitions. The syntax merger will deal with the cycles, and the type-checker will resolve ambiguities. The parser generator and parser of Fig. 1 need no adaptation.

4 Challenge 1: Cyclic derivations

We must remove all cycles that have been introduced by the syntax transitions. The following analysis identifies and characterizes all of these cycles. Using the characterization we will be able to deal with them. Use Fig. 2 as illustration.

Lemma 1. *Take any acyclic grammar O that does not define or use a non-terminal $Tree$. Obtain O' by adding for every non-terminal $N \in O$ two productions: $N \rightarrow Tree$ and $Tree \rightarrow N$. $Tree$ is not a start non-terminal of O' . Every cyclic derivation in O' will contain a sub-path of the form: $X \rightarrow Tree, Tree \rightarrow Y$ where X and Y are non-terminals in O .*

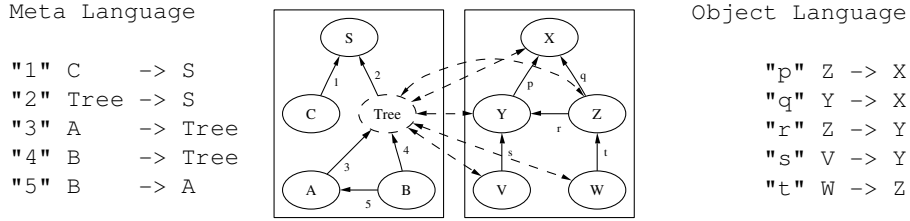


Fig. 2. A trivial meta syntax and object syntax are merged. Syntax transitions bidirectionally connect all non-terminals in the object language to the *Tree* non-terminal in the meta language

Proof. O is not cyclic, so at least one of the new productions is contained in every cycle in O' . Since *Tree* is not a start non-terminal, we can reach it only via a production of the form $\text{Tree} \rightarrow Y$. There are no other productions for *Tree*, so the only way to continue such derivations, including cyclic ones, is by a production of the form $X \rightarrow \text{Tree}$. This completes a cycle that contains the specified sub-path. \square

Lemma 2. Take O' from Lemma 1. Obtain $M(O')$ by adding to O' any grammar M that is acyclic. Assume that the only common non-terminal between M and O' is *Tree* and *Tree* is not a start non-terminal. Also assume *Tree* is reachable in M . $M(O')$ contains all cycles of O' , but no more.

Proof. Trivially, $M(O')$ contains at least all cycles from O' . Any extra cycles in $M(O')$ must use *Tree*, since M and O do not contain cycles and *Tree* is the only connection between M and O' .

A cycle that starts from a non-terminal in O' can reach *Tree* only by one of the generated productions $\text{Tree} \rightarrow Y$. Suppose the cycle can be completed by not immediately using a $Y \rightarrow \text{Tree}$ production back to O' . Then, eventually it must use some $X \rightarrow \text{Tree}$ production, otherwise M would have been cyclic. When a path in between these two productions completes a cycle, it must be a cycle in M too. This contradicts M being acyclic.

A cycle that starts from M can reach *Tree* in a number of ways. M is acyclic, so a cycle must contain at least one of the productions $\text{Tree} \rightarrow X$. Since X is defined to be a non-terminal in O , every cycle now completed from X would have been a cycle in the previous case (cycles starting in O').

So, no more cycles are produced by $M(O')$ than the ones that we already identified in O' . \square

Theorem 1. Take $M(O')$ from Lemma 2. Remove from the derivations produced by $M(O')$ all derivations containing a sub-path of the form $X \rightarrow \text{Tree}$, $\text{Tree} \rightarrow Y$. X and Y are any non-terminal from O . The resulting set of derivations is cycle-free.

Proof. By Lemma 1 all cycles in O' contain one of the designated sub-paths and by Lemma 2 $M(O')$ contains the same cycles as O' . Thus, removing these sub-paths from all derivations produced by $M(O')$ will break all cycles. \square

So, we can prevent all cycles in a merged syntax by removing certain sub-paths of derivations. We have assumed only that the meta syntax and the object syntax are

acyclic and they share no non-terminals. The `Tree` non-terminal should not be a start non-terminal.

In [6, 15] a method for removing unwanted derivations at parse-table generation time is described. This method can be used to implement relative operator priorities and associativity in expression languages in an efficient manner. For example:

```
Exp "*" Exp -> Exp > Exp "+" Exp -> Exp
```

This priority declaration in SDF removes all derivations where the "+" is a direct child of a "*" in a derivation. When parsing "1+2*3", the derivation that represents "(1+2)*3" is not available in the parse-table, leaving the parser to produce the only other possible derivation that represents "1+(2*3)".

We use this mechanism to remove all derivations $X \rightarrow Tree$, $Tree \rightarrow Y$ from the parse-table. For every combination of two non-terminals X and Y from the object syntax, we generate the following priority:

```
Tree -> Y > X -> Tree
```

These priorities remove all derivations where $X \rightarrow Tree$ is a direct child of $Tree \rightarrow Y$. By Theorem 1 this cuts off all cyclic derivation paths in the merged syntax. The resulting parse-table is not able to lead to any cyclic derivation during parsing.

The number of priorities generated is quadratic in the amount of non-terminals of the object syntax. Note that actual cycles only occur if there is a path from Y to X in the object grammar that does not use any terminal symbols. Such a path consists of injections and quasi-injections only. As an optimization that improves parse-table generation time, this information can be used to limit the amount of generated priorities to a less than quadratic number.

5 Challenge 2: Disambiguation

Classification of ambiguities. We assume that the meta grammar is not ambiguous and that the object grammar can be ambiguous. The syntax transitions possibly introduce new ambiguities. The complete syntax produced by the syntax merger may be ambiguous in the following ways:

Class 0: The object syntax is ambiguous by itself.

Class 1: Some object syntax non-terminals overlap, making `Tree` ambiguous.

Class 2: The meta language overlaps with the object language.

The first class of ambiguities should be left alone. We must not remove any of those ambiguities because they are not an artefact of the syntax merger, and may very well be intentional. The second class of ambiguities are prevented in other systems by explicit typing or integrating type-checking with parsing (Section 2). Instead, we will integrate disambiguation with type-checking. Note that this technique is also applicable for parsing concrete syntax with quotation to make some of the explicit typing unnecessary. The third class of ambiguities does not occur at all in systems that use quoting and anti-quoting. The quoting and anti-quoting operators effectively shield off such ambiguities

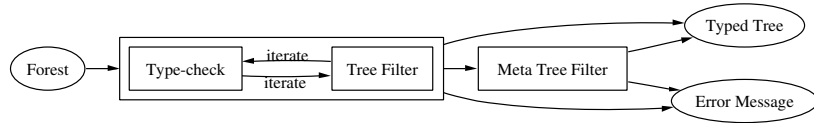


Fig. 3. The internal organization of the type-checking and disambiguation tool.

because they are so-called reserved keywords. We will extend the type-checker with *meta disambiguation rules* to deal with this class of ambiguities.

Figure 3 shows the internal organization of the type-checking and disambiguation tool. Note that if ambiguities of Class 0 exist, the resulting typed tree will be a forest containing only Class 0 ambiguities.

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 .r 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 [2]. Other approaches incorporate type-checking in the parsing phase [1, 13] 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. This approach follows the ideas of disambiguation by filtering as described in [9].

Type-checking forests is a natural extension of normal type-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 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; leave the disambiguation to the meta disambiguation phase.

Memory consumption of the abstract syntax forest is an issue. Abstract syntax forests have a high degree of redundancy, so we advise the use of maximal sub-term sharing [4]. Forests that do not use this technique tend to grow very rapidly in size. To optimize run-time behavior the type-checker should also to cache the result of type-checking every sub-tree. In the presence of such redundancy this is will ensure no double computation is done.

Note that resolving the ambiguities caused by syntax transitions by type-checking is a specific case of type-inference for polymorphic functions [12]. 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.

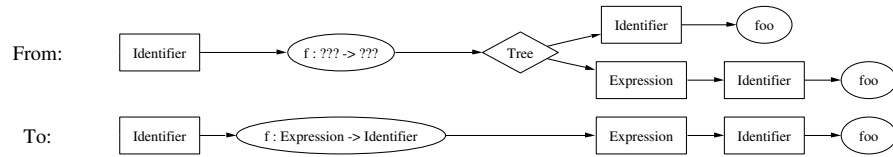


Fig. 4. An abstract syntax forest is disambiguated by using a type declaration for the function f .

Class 1: Ambiguous syntax transitions. There are two sub-classes of ambiguities we have to deal with:

Class 1.1: Several $X \rightarrow \text{Tree}$ are possible from different X s.

Class 1.2: Several $\text{Tree} \rightarrow X$ are possible to different X s.

The $X \rightarrow \text{Tree}$ productions allow everything in the object syntax to be Tree . By this production we transit from the meta grammar into the object grammar. So, the immediate typing context is provided by the meta language. Disambiguation is obtained by selecting the tree that has the $X \rightarrow \text{Tree}$ production on top where the X non-terminal corresponds to the type that the context expects.

The example in Fig. 4 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 .

The $\text{Tree} \rightarrow X$ productions cause that at any part of the object language can contain a piece of meta language. We transit from the object grammar into the meta grammar. The only pieces of meta language allowed are produced by the Tree non-terminal. The typing context is now provided by the object language. To disambiguate we select the meta expression that can be assigned type X .

To combine the above two cases 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. The more decisions can be made on the way down, cutting off branches before they are traversed, the more efficient the type-checking algorithm will be.

It is unlikely, but certainly possible that after this detailed type analysis ambiguities still remain. The type-checker can produce an accurate error message in that case. The approach assumes all program fragments are located in a typing context. Language that do not have such an assumption use either explicit typing for the top of a fragment, or provide a meta disambiguation rule as described in the following.

Class 2: Meta syntax overlaps with object syntax. 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 Class 2. The meta language is ASF+SDF, and the object language is Java:

```

[ ] int[ ] foo = int[ ]
[ ] foo = bar
or
[ ] int[ ] foo = int[ ][ ] foo = bar

```

The parser returns two alternative interpretations. The first has two equations, the second only one. By using suggestive layout we demonstrate 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. By applying this phase after type-checking we ensure that only type-correct alternatives are subject to this phase (Fig. 3). A number of implicit but obvious rules will provide a full separation between meta and object language. The rules are not universal, ASF+SDF may choose a different design than TXL. The design space is limited to making a number of choices:

- Rule 1:** Prefer declared meta identifiers over object identifiers, or vice versa.
- Rule 2:** Maximize or minimize the length of the path of injections and quasi-injections from the object syntax that starts from the `Tree` non-terminal and ends with a production that is not an injection or a quasi-injection.
- Rule 3:** Maximize or minimize the number of meta language productions used in a derivation.
- Rule 4:** Propose explicit quoting versus propose a rephrasing that will allow the other rules to disambiguate.

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 enough. This can happen in systems with polymorphic functions for example, or in the total absence of a typing context. Choosing the shortest path is a reasonable choice for most systems.

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

The above rules should be tried in order. Rule 1 is an essential rule for meta programming without syntactic hedges, it will fire frequently. Rule 2 is needed when a single type-context is not available. The other rules ensure full disambiguation in the rare cases where syntactic overlap coincides with type correct alternatives and Rule 1 does not apply. A warning message to the programmer in case Rule 3 or 4 fires is preferable since these cases rarely occur and are therefore unexpected.

After the type-checking and meta disambiguation phases all ambiguities introduced by the syntax merger have been identified. Only type-correct alternatives remain after type-checking, then the first two meta rules take care of basic "quote inference" when necessary. The last two rules are fail-safes for degenerate cases.

6 Experience

This work has been applied to parsing ASF+SDF specifications. First ASF+SDF was extended with function syntax to make the meta language more complex. Before there were only meta variables in ASF+SDF. Then a prototype syntax merger was developed that generates priorities for all combinations of non-terminals in the object syntax. As predicted, all cycles were removed by the generated priorities.

Both experience with post-parsing disambiguation filters in ASF+SDF [5], and the efficient implementation of type-inference algorithms for languages as Haskell and ML suggests that our type-checking disambiguation phase can be implemented efficiently. A full implementation of our architecture will provide a definitive analysis on the efficiency of parsing and type-checking real meta programs in this manner. We will focus on analyzing the run-time behavior of the SGLR algorithm in this environment.

7 Conclusion

A straightforward architecture for parsing meta programs with concrete syntax is presented. By using implicit transitions syntactic hedges are not necessary. This results in more readable meta programs.

The technical consequences of having implicit transitions, cycles and ambiguities, are solved in a three-tier architecture: syntax merging, parsing, and type-checking. The syntax merger adds priorities to let cyclic derivations be filtered from the parse-table. Then the SGLR algorithm produces a forest of trees. Next, using type-inference techniques type-correct trees are filtered. Finally, a small set of disambiguation rules takes care of making the final separation between meta and object language.

The resulting architecture shows strong separation of concerns. Also, It can be applied to meta programming languages with either simple or complex type systems, and can provide the user of such systems with clear error messages.

Future work entails a full implementation of the described architecture and integrating it in the programming environment of ASF+SDF. This will provide better error messages and make the type-system of the ASF+SDF type-system easily extensible. Another direction of future work is to add concrete syntax features to other meta programming languages, such as Java-based meta programming tools.

Acknowledgments. The author would like to thank Paul Klint, Mark van den Brand and Tijs van der Storm for their valuable comments on drafts of this paper.

References

1. A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects in functional languages. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 96–105. ACM Press, 1988.
2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

3. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.
4. 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.
5. M.G.J. van den Brand, S. Klusener, L. Moonen, and J.J. Vinju. Generalized Parsing and Term Rewriting - Semantics Directed Disambiguation. In B. Bryant and J. Saraiva, editors, *LDTA'03*, volume 82 of *ENTCS*. Elsevier, 2003.
6. 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, *CC'02*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.
7. J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.
8. J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
9. 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, 1994. Tech. Rep. 126, Dipartimento di Scienze dell'Informazione, Università di Milano.
10. B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, 1966.
11. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Mass., 1966.
12. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
13. J. Paakki. Attribute grammar paradigms: a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.
14. M.P.A. Sellink and C. Verhoef. Native patterns. In M.R. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society Press, 1998.
15. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
16. 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.
17. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 299–315. Springer-Verlag, 2002.