Generalized Type-Based Disambiguation of Concrete Object Syntax

3

ABSTRACT

In metaprogramming with concrete object syntax, object-level programs are composed from fragments written in concrete syntax. The use of small program fragments in such quotations and the use of meta-level expressions within these fragments (anti-quotation) often leads to ambiguities. This problem is usually solved through explicit disambiguation, resulting in considerable syntactic overhead. A few systems manage to reduce this overhead by using type information during parsing. Since this is hard to achieve with traditional parsing technology, these systems provide specific combinations of meta and object languages, and their implementations are difficult to reuse.

In this chapter, we generalize these approaches and present a *language independent* method for introducing concrete object syntax without requiring explicit disambiguation. The method uses scannerless generalized LR parsing to parse meta programs with embedded object-level fragments, which produces a forest of all possible parses. This forest is reduced to a tree by a disambiguating type-checker for the metalanguage. To validate our method we have developed embeddings of several object languages in Java, including AspectJ and Java itself.

3.1 INTRODUCTION

Meta-level programs analyze, transform, and generate *object-level* programs. It is commonly agreed that such program manipulations are best carried out on a structured representation of the object program in order to achieve compositionality of transformations and to guarantee well-formedness of the resulting program. Furthermore, structured representations support type safety and hygiene more easily. However, the notation for structured representations is usually verbose and rather different from the notations of the language under consideration, rendering it impractical as a syntax for object programs. Using the concrete syntax of the object language as a notation for this structured representation provides the best of both worlds. The metaprogram can be written using the concise, well-known syntax of the object language, while the underlying representation is still structured.

Syntactically checked concrete object syntax is now available in many metaprogramming systems. Syntax macro systems such as <bigwig> [Brabrand & Schwartzbach 2002], code generators such as Jak (JTS/AHEAD) [Batory et al. 1998] and Meta-AspectJ (MAJ) [Zook et al. 2004], and program transformation

systems such as ASF+SDF [van Deursen et al. 1996], DMS [Baxter et al. 2004], Stratego/XT [Visser 2002] and TXL [Cordy et al. 1991] all provide concrete object syntax. Some of these systems are designed for a specific object language, others are configurable for different object languages. In [Visser 2002] we presented a general architecture for introducing concrete syntax for any object language in any metalanguage. The approach employs modular syntax definition in SDF and Scannerless Generalized LR (SGLR) parsing for defining the syntax and parsing the combined meta and object language [Visser 1997b, van den Brand et al. 2002].

A remaining problem of concrete object syntax is that the syntax of the combined meta and object languages is usually highly ambiguous if the object language is embedded using a single pair of quotation and anti-quotation symbols. Most systems solve this by using a different quotation and anti-quotation symbol for each nonterminal of the object language, leading to considerable syntactic clutter and requiring the metaprogrammer to be intimately familiar with the syntactic structure of the object language. Because of the irregularity of the embedding, the set of syntactic categories that can be quoted and unquoted is usually limited. Moreover, in a language with manifest typing that already requires programmers to declare the types of all variables, the disambiguation of quotations feels redundant. For example, consider the following fragment written in Jak (part of the JTS/AHEAD Tool Suite [Batory et al. 1998]):

```
Stmt stm2 = stm{ if($exp(e1)) { $stm(stm1); }; }stm;
```

Here a statement *stm2* is constructed from an expression *e1* and a statement *stm1*. The syntactic categories of the quotation `stm{...}stm` and the antiquotations `$exp(`*e1*`)` and `$stm(`*stm1*`)` within it are explicitly indicated using the identifiers `stm` and `exp`.

Meta-AspectJ (MAJ) [Zook et al. 2004], an extension of Java for the generation of AspectJ programs, reduces the need for different quotation and anti-quotation symbols by means of a context-sensitive parser, taking variable declarations into account during parsing. For example, in MAJ the Jak fragment above can be written as follows:

```
Stm stm2 = `[ if(#e1) { #stm1 } ]
```

The syntactic categories of the fragment and the variables are inferred from the explicit declaration of their types in the program. Thus, MAJ requires from the programmer less knowledge of the embedding and the syntactic details of the object language. However, the implementation of MAJ is specific to the embedding of AspectJ in Java, and is not easily reusable for embeddings of other languages, due to a number of limitations. First, the scanner for meta and object language is the same, which precludes embedding of languages with a different lexical syntax. Second, it is not possible to extend the metalanguage with concrete object syntax for *multiple* languages, since the implementations of context-sensitive parsing do not compose. Finally, the implementation of parsing and type-checking is tangled, which leads to complex

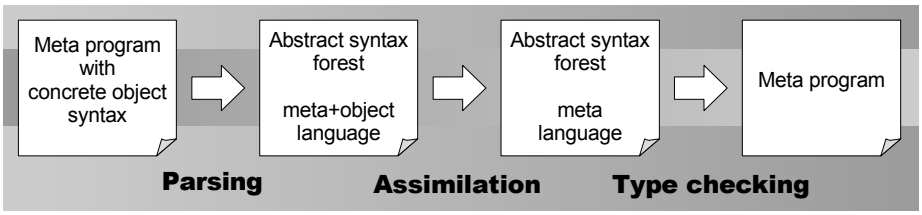

Figure 3.1 Architecture of generalized type-based disambiguation.

and hard to maintain code that has limitations that might surprise users. For example, MAJ cannot always handle overloaded methods that are invoked with quoted arguments.

In this chapter, we describe an extension of our general architecture for concrete object syntax with type-based disambiguation that allows embeddings with minimal syntactic overhead. The main characteristic of our approach is that ambiguities are preserved by the parser and are solved in a separate phase by an extension of a type-checker that operates on an abstract syntax forest. This separation of phases is illustrated in Figure 3.1. As a result, language embedding and assimilation (expansion of embedded object code to the metalanguage) can remain compositional. Therefore, it is easy to add new object languages and to combine object language embeddings. Since ambiguities are solved *after* assimilation, the implementation of disambiguation for a metalanguage is *object language independent*. However, we require that the representation of object programs in the metalanguage is typed and that distinct syntactic categories have a different type in this representation (see Section 3.4.4 and 3.5.2). Disambiguation is achieved by a natural and orthogonal extension of the type system. By separating the issue of disambiguation from the type-checker, we can handle ambiguities in complex typing situations for free, hence reducing the number of exceptions and heuristics. Also, the approach is *not restricted to a single metalanguage*. Disambiguation is implemented as an extension of the type checker of the metalanguage, so the method is restricted to statically typed metalanguages, and not applicable to untyped languages. Furthermore, the method is particularly suitable (and desirable) for languages that use manifest typing (e.g. C, Java, C#). We have no experience with metalanguages using type inference.

We proceed as follows. In the next section we recapitulate the embedding and assimilation of an object language in a metalanguage. In Section 3.3 we examine the ambiguities caused by such embeddings and previous solutions used for them. In Section 3.4 we present a generalized type-based disambiguation method for concrete object syntax. In Section 3.5 we describe our experience with the method in a generic disambiguation implementation for Java as a metalanguage with embeddings of AspectJ and Java itself. In Section 3.6 we discuss previous, related, and future work.

```
37 module JavaJava
38 imports Java-15-Prefixed Java-15
39 exports
40   context-free syntax
41     "⟦" Expr "⟧"      -> MetaExpr {cons("ToMetaExpr")}
42     "#[" MetaExpr "]" -> Expr     {cons("FromMetaExpr")}
```

Figure 3.2 Syntax definition for simple embedding of Java expressions in Java

3.2 METAPROGRAMMING WITH CONCRETE OBJECT SYNTAX

In this section, we recapitulate the general method for adding support for concrete object syntax to a metalanguage, which was presented in [Visser 2002, Bravenboer & Visser 2004 (Chapter 2)]. Introduction of concrete object syntax in a metalanguage requires (1) embedding the syntax of the object language in the meta language and (2) assimilation of the embedded object code fragments to the metalanguage, expressed in terms of the underlying structured representation. The generality of the approach is based on syntax definition in the modular syntax definition formalism SDF for defining the embedding and the transformation language Stratego for the assimilation. We illustrate the approach with the introduction of concrete syntax for Java in Java.

3.2.1 *Embedding*

The embedding of an object language in a metalanguage requires the combination of syntax definitions for both languages. From this combined syntax definition a parser is generated, which is used to parse metaprograms that use concrete object syntax. Thus, the embedding of Java in Java is achieved by the module in Figure 3.2. The module imports [38] the Java syntax twice; once as the metalanguage (`Java-15-prefixed`) and once as the object language (`Java-15`). To avoid confusion between the two languages (or language roles in this case), the module `Java-15-prefixed` prefixes the nonterminals of the metalanguage with '`Meta`' by renaming them in its import declaration of `Java-15`.

Next, to actually integrate the meta and object language, the combination of these syntax definitions is extended with productions that determine the possible transitions from the metalanguage to the object language (quotation) and vice versa (anti-quotation). A *quotation* quotes a fragment of an object-level program and embeds it in a meta-level program. The first production [41] in Figure 3.2 defines that an object-level `Expr` between ⟦ and ⟧ can be used as a meta-level `MetaExpr`. The `cons` annotation in the production declares the constructor to be used in the abstract syntax tree. The following Java statement illustrates the quotation of a Java method call:

```
Expression x = ⟦ resultSet.getInt(4) ⟧
```

The meaning of this statement is Java code for the construction of the abstract syntax tree corresponding to the quoted fragment.

An *anti-quotation* is an escape from a quotation to the meta-level, to splice in pieces of object code computed elsewhere. The second production[42] in Figure 3.2 declares that a `MetaExpr` between `#[` and `]` can be used as an object-level `Expr`. For example, in the following quotation the method argument is an expression *foreignkey* that is determined from some domain specification:

```
Expression x = ⟦ resultSet.getInt(#[ foreignkey ]) ⟧;
```

3.2.2 *Assimilation*

Assimilation transforms a program with embedded object code to a pure meta-level program by translating the embedded fragments to code in the metalanguage that constructs the underlying abstract syntax tree representation. For example, in our Java in Java embedding we use the Eclipse JDT Core DOM [JDT Website] for representing the object programs. Hence, the Java constructs must be translated to invocations of the methods in this API. The following Stratego rewrite rules illustrate the assimilation for some Java language constructs. The first rule translates a return statement, the second rule a method invocation. The Stratego rewrite rules use concrete object syntax as well.

```
Assimilate(rec) :
  ⟦ return; ⟧ -> ⟦ _ast.newReturnStatement() ⟧

Assimilate(rec) :
  ⟦ e.y(e*) ⟧ ->
  ⟦ {| MethodInvocation x = _ast.newMethodInvocation();
       x.setName(~e:<AssimilateId(rec)> y);
       x.setExpression(~e:<rec> e);
       bstm*
     | x |}
  ⟧
  where <newname> "inv" => x
      ; <AssimilateArgs(rec | x)> e* => bstm*
```

In the assimilation rules we use a small extension `{|stmt*|expr|}` of Java, called an eblock, that allows the inclusion of statements in expressions. The value of an eblock is the expression. In the assimilation rules, the *italic* identifiers (e.g. *e*, *y*, and *e**) indicate meta-level variables, a convention we use in all the code examples. `~e:` denotes an anti-quotation where the result is a Java expression. `<s> p` applies the rewriting *s* to the pattern *p*. `s => p` matches the result of *s* to *p*. `newname` creates a fresh, unique, name, which guarantees hygiene in the assimilation. `AssimilateArgs` is a helper strategy that assimilates a list of expressions to arguments of the method invocation.

As an example, consider the result of assimilating the last example above, which illustrates the advantage of concrete syntax.

```
MethodInvocation inv = _ast.newMethodInvocation();
inv.setName(_ast.newSimpleName("getInt"));
inv.setExpression(_ast.newSimpleName("resultSet"));
List<Expression> args = inv.arguments();
args.add(foreignkey);
Expression x = inv;
```

```
1  "⟦" CompilationUnit "⟧" -> MetaExpr {cons("ToMetaExpr")}
2  "⟦" TypeDec         "⟧" -> MetaExpr {cons("ToMetaExpr")}
3  "⟦" BlockStm        "⟧" -> MetaExpr {cons("ToMetaExpr")}
4  "⟦" BlockStm*       "⟧" -> MetaExpr {cons("ToMetaExpr")}

5  "#[" MetaExpr "]"-> ID   {cons("FromMetaExpr")}
6  "#[" MetaExpr "]"-> Expr {cons("FromMetaExpr")}
```

Figure 3.3 Syntax definition for embedding of Java in Java

In the examples of this chapter, the assimilation is embedding specific, since the mapping of the object language to an existing API is inherently embedding specific. However, if there is a fixed correspondence between the syntax definition and the API, then the assimilation can be generic. This is typically the case if the API is generated from the syntax definition using an API generator such as ApiGen [van den Brand et al. 2005].

3.3 AMBIGUITY IN CONCRETE OBJECT SYNTAX

In this section we discuss how ambiguities can arise when using concrete object syntax. Also, we discuss how these ambiguities are handled in related work.

3.3.1 *Causes of Ambiguity*

Lexical State If a separate lexical analysis phase is used to parse a metaprogram, then ambiguities will arise if the lexical syntax of the object language is different from the metalanguage. The set of tokens of both languages cannot just be combined, since the tokens of both languages are only allowed in certain contexts of the source file. For example, `pointcut` is a keyword in embedded AspectJ, but should not be in the surrounding Java code.

Quotation Ambiguous quotations can occur if the same quotation symbols are used for different nonterminals of the object language. If the object code fragment in the quotation can be parsed with both nonterminals, then the quotation itself is ambiguous as well. For example, consider the SDF productions [1] and [2] in Figure 3.3 that define a quotation for a compilation unit and a type declaration. With these two quotation rules, the fragment ⟦ `class Foo { }` ⟧ is ambiguous, since the quoted Java fragment can be parsed as a compilation unit as well as a type declaration. Note that not all quotations are ambiguous: if the object code includes a package declaration or imports, then it cannot be parsed as a type declaration. A similar ambiguity issue occurs if the embedding allows quotation of lists of nonterminals as well as single nonterminals. For example, consider the SDF productions [3] and [4] in Figure 3.3 for quoting block statements. A quotation containing a single statement is now ambiguous, since it can be parsed using both production rules.

Anti-Quotation Similar ambiguity problems occur when using the same anti-quotation symbols for different nonterminals of the object language. For example, consider the anti-quotations [5] and [6] in Figure 3.3 for identifiers and expressions. The anti-quotation in `⟦ #[a] + 3 ⟧` is ambiguous, since `#[a]` can represent an identifier as well as a complete expression.

3.3.2 *Solutions*

Lexical State Most systems use a separate scanner. The consequence is that the lexical analysis must consider lexical states and will often assume fixed quotation symbols to determine the current state. Alternatively, the scanner can interact with the parser to support a more general determination of the lexical state. Some other systems just take the union of the lexical syntax, hence forbidding reserved keywords of the object language in the metalanguage. MAJ also reserves several keywords to work around lexical ambiguities (e.g. `pointcut` is a meta keyword) and some of these keywords are not even part of the object language (e.g. `VarDec` and `args`). ASF+SDF and Stratego both use *scannerless parsing* for parsing meta programs. Lexical ambiguities are not an issue in scannerless parsing, since they inherently only occur if a separate scanner is used.

Explicit Typing Ambiguous quotations and anti-quotations can be solved by requiring explicit disambiguation by using different quotation symbols. For example, JTS uses different quotations for the class example: `prg{...}prg` for compilation units and `cls{...}cls` for class declarations. Stratego uses the same solution, but the disambiguated versions of the quotations are optional: if there is no ambiguity, then the general quotation symbols can be used. For example, `⟦ package foo; class Foo {} ⟧` is not ambiguous (it is a compilation unit), but a plain class declaration requires an explicit disambiguation, e.g. `compilation-unit ⟦ class Foo {} ⟧`.

JTS solves ambiguities between quotations of a single nonterminal and a list of nonterminals in two different ways. First, there are specific quotations for lists, for example `xlst{...}xlst` for the arguments of a method call. Second, some nonterminals only have a single quotation instead of two, where this single quotation always represents a list. In Stratego, list quotations are explicitly disambiguated, e.g. `bstm* ⟦ x = 5; ⟧`.

Context-sensitive Parsing MAJ uses context-sensitive parsing to solve ambiguous quotations and anti-quotations, by using type information at parse-time to infer the type of the quotation or anti-quotation to be parsed. Concerning list quotations, if `infer` is used, MAJ uses a single element if possible and an array if it must be a list. If the type of the variable is declared, then this type is considered. For example, the quotation in the statement `Stmt[] stms = ‘[ x = 4; ];` will be parsed to the construction of an array instead of a single statement. Hence, explicit disambiguation of the quotation itself is not necessary. Unfortunately, MAJ does not implement full support for the type system of Java and uses common interfaces for conceptually different

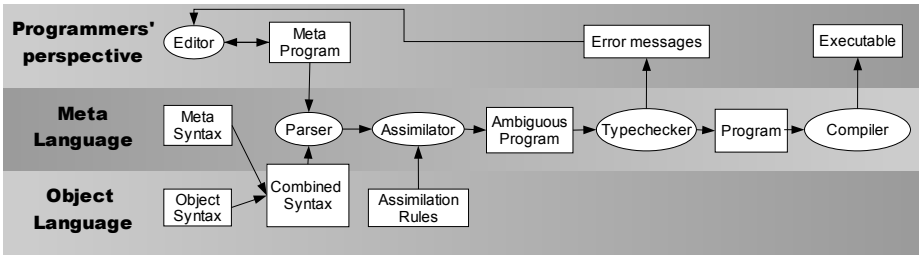

Figure 3.4 Architecture of embedding and assimilation framework with type-based disambiguation.

AST classes to work around issues in the quotation inference. Section 3.5.2 discusses these problems in more detail.

Grammar Specialization ASF+SDF is a system with first order types. It translates this type system to a context-free grammar, thus parsers can be generated that accept only type correct metaprograms. As a result, neither quoting of object fragments, nor anti-quoting of metavariables, nor explicit typing is necessary in ASF+SDF. However, the type system is limited to first order types only. Remaining ambiguities are currently solved by using heuristic disambiguation filters, such as injection count. In [Vinju 2005] a type-based solution for these ambiguities is presented, where grammar generation is no longer necessary.

3.4 GENERALIZED TYPE-BASED DISAMBIGUATION

In summary, quotation and anti-quotation can be used to introduce concrete syntax for object-level program fragments, but need some form of disambiguation. Explicit disambiguation methods introduce syntactic clutter that obscures metaprograms. Reduction of this syntactic clutter can be achieved by using type information for disambiguation. While MAJ does a great job at achieving this for the specific embedding of AspectJ in Java, its implementation is hard to generalize to other object languages and to the combination of multiple object languages, because of the poor compositionality of its context-sensitive parsing algorithm.

In this section, we introduce an alternative approach that generalizes easily to arbitrary object languages. Indeed it is generic in the embedded object language and can easily be transposed to other metalanguages, considering the restrictions on the type system, as mentioned in the introduction. We illustrate the method with the embedding of Java in Java, but stress that the architecture and implementation is object language independent. The basic idea of the approach is to perform *type-based disambiguation of an abstract syntax* forest *after assimilation*. The architecture of our method is illustrated in Figure 3.4. In the rest of this section we describe the elements of the pipeline.

3.4.1 *Syntax Definition and Parsing*

The first stage of the pipeline consists of parsing the metaprogram with a parser generated from the combined syntax definition. This phase preserves all the ambiguities in the metaprogram, by employing generalized LR parsing. The result is a parse *forest*, that is, a compact representation of all possible parses of the program. At points where multiple parses are possible the forest contains *ambiguity nodes* consisting of a set of all alternative parse trees, or in fact forests, since ambiguities can be nested. As a technical note, we actually consider *abstract syntax forests*, that is parse forests with irrelevant information such as whitespace, comments, and literals removed. For example, the Java assignment statement `dec = ⟦ class Foo {} ⟧;` is parsed to the following abstract syntax forest in term notation (where we have elided some details of the structure of class declarations having to do with modifiers and such):

```
Assign(
  ExprName(Id("dec"))
, 1> ToMetaExpr(CompilationUnit(... ClassDec(... Id("Foo") ...) ...))
  2> ToMetaExpr(ClassDec(... Id("Foo") ...))
  3> ToMetaExpr([ ClassDec(... Id("Foo") ...) ])
  4> ToMetaExpr(ClassDecStm(ClassDec(... Id("Foo") ...)))
  5> ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...)) ])
)
```

In this forest it is clear that the right-hand side of the assignment is ambiguous and has five alternative parses. We use the notation `1>...n>` to indicate the alternatives of an ambiguity node. The five alternatives are a compilation unit containing a class declaration, a class declaration on its own, a singleton list of a class body declaration declaring a member class (see Section 3.3.1 for a discussion of ambiguities caused by lists), a block statement of a class declaration local to a method, and a singleton list of the same block statement. The `ToMetaExpr` constructor represents a transition from the metalanguage to the object language (see Figure 3.3). To make the running example a bit more concise, we leave out the alternatives of the block statements (4 and 5). The first three alternatives are sufficient to illustrate all the issues.

3.4.2 *Assimilation*

The second stage in the pipeline is assimilation, i.e., the translation of the embedded language fragments to their implementation in the metalanguage as described in Section 3.2.2. The only difference is that assimilation now transforms a forest instead of a tree. If the assimilation rules are compositional (i.e. the transformed fragments are small) there is no interference between regular assimilation rules and ambiguities, that is, ambiguities are preserved during assimilation. Thus, after assimilation, the abstract syntax forest only contains metalanguage constructs and ambiguity nodes. For example, the following code fragment shows the intermediate result after assimilation of the example above to the Eclipse JDT Core DOM (again some details have been elided).

```
1> {| CompilationUnit cu0 = _ast.newCompilationUnit();
      List<AbstractTypeDeclaration> args1 = cu0.types();
      args1.add(
        {| TypeDeclaration class0 = _ast.newTypeDeclaration();
           class0.setName(_ast.newSimpleName("Foo"));
           ...
         | class0 |});
      ...
    | cu0 |}
2> {| TypeDeclaration class1 = _ast.newTypeDeclaration();
      class1.setName(_ast.newSimpleName("Foo"));
      ...
    | class1 |}
3> {| List<BodyDeclaration> decs0 = new ArrayList<BodyDeclaration>();
      decs0.add(
        {| TypeDeclaration class2 = _ast.newTypeDeclaration();
           class2.setName(_ast.newSimpleName("Foo"));
           ...
         | class2 |});
      ...
    | decs0 |}
```

3.4.3 *Type-Based Disambiguation*

In the final stage of processing the metaprogram, ambiguities are resolved. The disambiguation operates on an abstract syntax forest of the metalanguage without any traces of the object language. Thus, the disambiguation phase does not have to be aware of quotations and anti-quotations, or of their contents. The disambiguation is implemented as an extension of a type-checker for the metalanguage that analyses the abstract syntax forest and eliminates the alternatives that are not type correct. The algorithm for disambiguation is sketched in Figure 3.5. From within the type-checker the disambiguate function is invoked for every node *node* in the abstract syntax forest after typing it.

The disambiguate function distinguishes three cases, which we discuss in reverse order. If the node *node* is not ambiguous it is just returned. If one of the sub-nodes of *node* is ambiguous, its alternatives are lifted to the current node by lift-ambiguity. Its definition states that if n is equal to some ambiguity node within a context `c[.]`, the context is distributed over the ambiguity (We give an example of distribution of an assignment shortly). Finally, if the node *node* is directly ambiguous or after lifting the ambiguities from its sub-nodes, the resolve function is used to resolve the ambiguity.

The resolve function takes an ambiguous node and removes from it all alternatives that are not type correct. This may result in an empty set of alternatives (*#node'* = 0), which indicates a type error, a singleton set (*#node'* = 1), which indicates that the ambiguity is solved, or a set with more than one alternative (*#node'* > 1). In the latter case, if the ambiguity involves a statement or declaration no more context information can be used to select the intended alternative, hence it is reported as an ambiguity error. Otherwise, in the case of an expression, the ambiguity is allowed to be lifted into its parent level, where it may be resolved due to context information.

function disambiguate(*node*) =

```
1  if node is ambiguous then
2    return resolve(node)
3  else if node has ambiguous child then
4    return resolve(lift-ambiguity(node))
5  else
6    return node
```

function resolve(*node*) =

```
1  node' := remove from node all alternatives which are not type correct
2  if #node' = 0 then
3    report type error
4  else if #node' = 1 then
5    return node'
6  else if #node' > 1 then
7    if node' contains a meta statement or declaration then
8      report ambiguity error
9  else
10   return node'
```

function lift-ambiguity(*node*)

```
1  if node = c[ 1> node_1  2> node_2  ...  j> node_j ] then
2    return  1> c[node_1]  2> c[node_2]  ...  j> c[node_j]
```

Figure 3.5 Algorithm for type-based resolution of ambiguities

To illustrate the lifting and elimination of ambiguities, consider the ambiguity between the compilation unit, type declaration and list of body declarations in the assimilated example. If this ambiguous expression occurs in an assignment, i.e.

```
dec = 1> CompilationUnit 2> TypeDeclaration 3> List<BodyDeclaration>
```

then the ambiguity will be lifted out of the assignment (for brevity, the actual expression has been replaced by its type). This will result in a new ambiguity node with three alternatives for this assignment, i.e.

```
1> dec = CompilationUnit 2> dec = TypeDeclaration 3> dec = List<BodyDeclaration>
```

Depending on the type of the variable `dec`, two of these assignments will most likely be eliminated. For example, if the variable has type *TypeDeclaration*, then the *CompilationUnit* and *List<BodyDeclaration>* assignments will be eliminated, since these assignments cannot be typed. Note that this mechanism requires variables to be declared with a reasonably specific type. That is, if the variable `dec` has type `Object` then all the assignments can be typed and an ambiguity error will be reported.

Similarly, ambiguities are lifted out of method invocations: For example,

```
f(1> CompilationUnit 2> TypeDeclaration 3> List<BodyDeclaration>)
```

is lifted to

```
1> f(CompilationUnit) 2> f(TypeDeclaration) 3> f(List<BodyDeclaration>)
```

If `f` is just defined for one of these types, then just one of the invocations can be typed. Thus, the other invocations will be eliminated. On the other hand, if `f` is overloaded or defined for a supertype of two or more of the types, then the ambiguity will be preserved. It might be eliminated later, if the result types of `f` are different. If this is not the case, then an ambiguity will be reported, similar to an ambiguous method invocation in plain Java.

3.4.4 *Explicit Disambiguation*

For cases that are inherently ambiguous or just unclear, explicit disambiguation can be used. Most systems introduce special symbols for this purpose, but due to our integration in the type-checker one may use *casting* to inform the type-checker that something should a have certain type. The implementation of the explicit disambiguation comes for free, since incorrect casts cannot be type-checked. Thus, these alternatives will be eliminated. For example, in our running example a cast to a compilation unit `(CompilationUnit) ⟦ public class Foo {} ⟧` will cause the alternatives to be eliminated. In this way, any object language construct can be disambiguated, not only the ones that the developer of the embedding happens to support.

However, there is a situation where not even casting will help. Our method requires that the underlying structured representation of the object language is typed and that distinct syntactic categories in the object language have a different type in this representation. For example, if the structured representation is a universal data format such as XML or ATerms, then our method will not be able to disambiguate the concrete object syntax, since the different syntactic categories are not represented by different types in the metalanguage. Fortunately, a sufficiently typed representation is preferable anyway, since it would otherwise be possible to construct invalid abstract syntax trees. Note that similar problems occur in a dynamically typed languages. As mentioned in the introduction, our method is most suitable for statically typed languages.

3.5 EXPERIENCE

To exercise the general applicability of our method to the embedding of different object languages, we implemented two large embeddings. Small fragments of the first application have already been presented in several examples: the embedding of Java in Java using assimilation to the Eclipse JDT Core DOM. We call this embedding JavaJava. The second application embeds AspectJ in Java and mimics the object language specific implementation of MAJ. Although AspectJ is a superset of Java, these two applications are quite different, since the embedding of AspectJ assimilates to the MAJ abstract syntax tree. The applications substantiate our claims, but also give some interesting

insights in the limitations and the relation to object language specific implementations.

3.5.1 *JavaJava*

The implementation of JavaJava consists of a syntax definition (small fragment presented in Figure 3.3) and a set of assimilation rules that translate Java 5.0 abstract syntax tree constructs to the Eclipse JDT Core DOM [JDT Website] (examples have been shown in Section 3.2.2).

The mapping from the syntax definition to the Eclipse DOM is natural, since both are based on the Java Language Specification. Furthermore, the DOM is well-designed and uses distinct classes to represent distinct syntactic categories. Because of this, our type-based disambiguation works quite well for JavaJava.

An interesting issue is the type of containers used in the DOM. The DOM uses unparameterized standard Java collections, as opposed to arrays, or type specific containers. So although the DOM itself can *represent* parameterized types in an object program, the DOM implementation itself does not *use* parameterized types. Our disambiguating type-checker would benefit from parameterized collections, by harvesting the additional type information about the elements of a collection (e.g. `List<Expression>`). Fortunately, parameterized types and unparameterized types can be freely mixed, i.e. we can still use parameterized types in metaprograms. However, we prefer a more precisely typed DOM, such that unchecked conversions or explicit casts can be avoided. Note that this shows that a sufficiently typed representation is important for our method.

3.5.2 *Meta-AspectJ*

We developed the embedding of AspectJ in Java to compare our generalized and staged disambiguation solution to a specific implementation, namely MAJ. For this, we also had to study the behaviour of MAJ in more detail. Our syntactic embedding is based on a modular AspectJ and Java syntax definition in SDF and exactly mimics the syntax of MAJ. The syntactic embedding was very easy to implement using SDF: basically we just have to combine the existing syntax definitions in a new module. The syntax definition also supports the explicit disambiguations of MAJ, but these are not really necessary, since casts can be used in our embedding method. For the underlying structured representation we use the MAJ AST.

We learned that our generalized implementation of disambiguation in a separate type-checker has the advantage that it is much easier to implement support for more advanced Java constructs. For example, our implementation fully supports disambiguation of quotations in method invocations by performing complete method overload resolution, which MAJ does not. So, given the following overloaded method declarations:

```
Stmt     foo(CompilationUnit cu) { ... }
JavaExpr foo(ClassDec dec)       { ... }
```

our implementation can disambiguate invocations of the `foo` method that take quoted AspectJ code as an argument:

```
Stmt stmt     = foo(‘[ class MyClass {} ]);
JavaExpr expr = foo(‘[ class MyClass {} ]);
```

On the other hand, MAJ as an object language specific implementation provides some additional, object language specific, functionality that is not available in our generalized implementation. For instance, MAJ supports the conversion of Java (meta-level) values to AspectJ (object-level) expressions. For example, an array can be used as a variable initializer without converting it to the object-level by hand. Unfortunately, this conversion cannot be handled in a generic way, since it is not applicable to all object languages. However, for the specific embedding of Java in Java this could be added to the type-checker (see future work).

We have not implemented the `infer` feature of MAJ, which supports inferring the type of a local variable declaration. Hence, the types of all variables should be declared in our implementation. The `infer` feature itself is not hard to implement, but we would have to introduce heuristics to disambiguate ambiguous expressions, since no type is declared for the variable. MAJ applies such heuristics, for example by choosing a `ClassDec` if the type of the variable is `infer`, even if a `MajCompilationUnit` would also be possible. To work around incorrect choices, similar abstract syntax tree classes implement a common interface. For example, `ClassDec` and `MajCompilationUnit` implement the common interface `CompUnit`. This is a nice example of the problem mentioned in Section 3.4.4: distinct syntactic categories share a common interface. Thus, the declaration `CompUnit c = ‘[class Foo {}];` will result in an ambiguity error in our approach.

3.6 DISCUSSION

3.6.1 *Previous Work*

We use the modular syntax definition formalism SDF [Visser 1997b], with integrated lexical and context-free syntax and declarative syntactic disambiguation constructs. It is implemented using scannerless generalized LR parsing [Visser 1997b, van den Brand et al. 2002]. SDF is developed in the context of the ASF+SDF Meta-Environment [van Deursen et al. 1996], but is used in several other projects such as ELAN [van den Brand & Ringeissen 2000]. Our Stratego/XT [Visser 2004] program transformation system uses SDF for parsing, in particular of metaprograms with concrete object syntax [Visser 2002]. Stratego is not statically type-checked. Therefore, it employs quoting with explicit typing, where necessary.

The ASF+SDF Meta-Environment is a metaprogramming system based on term rewriting. It uses grammar specialization to resolve ambiguities caused by object language fragments. To let the type system of ASF+SDF deal with parametric polymorphism, in [Vinju 2005] a separate disambiguating type-

checker replaces the grammar generation scheme. The solution of [Vinju 2005] instantiates the framework described in this chapter for ASF+SDF.

Section 3.2 describes previous work on hosting arbitrary object languages in any host language [Bravenboer & Visser 2004 (Chapter 2)], which generalizes the approach taken for Stratego [Visser 2002] to any general purpose programming language. In the examples of [Bravenboer & Visser 2004 (Chapter 2)], Java was used as the host language and we also embedded Java as the object language in Java. However, explicit disambiguation was required and an untyped underlying representation was used. The contribution of generalized type-based disambiguation is the introduction of a disambiguating type-checker to remove the need for explicit typing. Moreover, the implementation is generic in the embedded object language. Thus, we obtain a similar notation as found in ASF+SDF, but can handle more than simple first order type systems, and use no disambiguation heuristics.

3.6.2 *Related Work*

The subject of embedding the syntax of object languages into host languages has a long history. The following discussion is meant to position our work more precisely.

Early work on syntactic embeddings revolves around the concept of syntax macros [Leavenworth 1966]. They allow a user to dynamically extend a general purpose programming language with syntactic abstractions. These abstractions are defined in programs themselves. Implementations of this idea have been limited to certain subclasses of grammars, like LL(1) and LALR, as an argument of a fixed macro invocation syntax. Thus, these approaches can not be transferred to our setting of hosting arbitrary object languages.

The work of Aasa [Aasa et al. 1988] in ML is strongly related to our setting. By merging the parsing and type-checking phases for ML, and using a generalized parsing algorithm, this system can cope with arbitrary context-free object languages. It uses a fixed set of quotation and anti-quotation symbols that allow explicit typing to let the user disambiguate in case the type system cannot decide. As opposed to this solution, our approach completely disentangles parsing from type-checking, and allows user defined quotation and anti-quotation symbols.

DMS [Baxter et al. 2004] and TXL [Cordy et al. 1991] are specialized metaprogramming environments similar to ASF+SDF and Stratego/XT. In DMS the user can define AST patterns using concrete syntax, which are quoted and guarded by explicit type declarations. TXL has an intuitive syntax with keywords that limit the scope of object code fragments, instead of quoting symbols that surround every code fragment. Each code fragment, and each first occurrence of a meta variable is explicitly annotated by a type in TXL.

The Jakarta Tool Suite (JTS) [Batory et al. 1998] and the Java Syntax Extender (JSE) [Bachrach & Playford 2001] are Java based solutions for metaprogramming and extensible syntax. Our framework, consisting of scannerless generalized LR parsing and type-based disambiguation, is more general than

the parsing techniques used by these systems. JTS uses explicit quotation and explicit typing, which can be avoided with our framework. Maya [Baker & Hsieh 2002] uses extensible LALR for providing extensible syntax. Multi-dispatch is used to allow multiple implementations of new syntax, where the alternatives have access to the types of the arguments. Unfortunately, a separate scanner and LALR limit the syntactic flexibility. MAJ [Zook et al. 2004] obtains type-based disambiguation for the embedding AspectJ in Java, using context-sensitive parsing. We contribute by disentangling the parser from the type-checker, resulting in an architecture that can handle any context-free object language. Our architecture stays closer to the original Java type system, in order to limit unexpected behavior.

Camlp4 [de Rauglaudre 2003] is a preprocessor for OCaml for the implementation of syntax extensions using an extensible top down recursive descent parser. New language constructs are translated to OCaml code by syntax expanders that are associated to the syntax extensions. Camlp4 provides quotations and anti-quotations to allow the generation of OCaml code using concrete syntax. The contents of quotations is passed as a string to a quotation expander, which can then process the string in arbitrary ways. A default quotation expander can be defined, but all other quotations have to be typed explicitly. The same holds for syntactically ambiguous anti-quotations. As opposed to Maya, the syntax and quotation expanders cannot use type information to decide what code to produce.

The method of disambiguation we use is an instance of a more general language design pattern called "disambiguation filters" [Klint & Visser 1994]. Although there are lightweight methods for filtering ambiguities that are very close to the syntactic level [van den Brand et al. 2002], disambiguation filters can generally not be expressed using context-free parsing. For example, any parser for the C language will use an extra symbol table to disambiguate C programs. Either more computational power is merged in parsers, or separate disambiguation filters are implemented on sets of parse forests [van den Brand et al. 2003]. We prefer the latter approach, because it untangles parsing from abstract syntax tree processing.

The problem of disambiguating embedded object languages is different from disambiguation issues in type-checkers, such as resolution of overloaded methods and operators. First, disambiguation in type-checkers can be done locally, based on the types of the arguments of the expression. Hence, lifting of the ambiguity, an essential part of our algorithm, is not used in such type-checkers. Second, in our approach large fragments of the program can be ambiguous and are represented in completely different ways. For typical ambiguities in programming languages, such as overloaded operators, the alternatives can conveniently be expressed in a single tree.

3.6.3 *Future Work*

We are considering to widen the scope of the framework in two directions. First, we would like to experiment with languages that have other kinds of

type systems, such as languages with type inferencing and languages with union types. Second, the assimilation of an embedded domain-specific language (beyond object languages) often requires more complex transformations of the metaprogram and the object fragments. Applying type-based disambiguation after assimilation is a problem in this case. Extending the type-checker of the host language with object language specific functionality is one of the options to investigate for this purpose. Also, the disambiguating type-checker might be interesting for other applications than disambiguation of embedded concrete object syntax. For example, it could be used for the reclassification of ambiguous names, or even to perform overloading resolution.

Object Language Specific Conversions

Many programming languages define conversions between types that can be applied in assignments, function calls, etc. For example, Java defines several narrowing, widening, boxing and unboxing conversions. In a certain way, quotations can be used to perform additional, object language specific conversions. For example, an identifier can be converted to an expression by just quoting it. Similarly, a type declaration can be converted to a compilation unit.

```
String s = ...;
Expression e = |[ #[x] ]|;
TypeDeclaration classdec = ...;
CompilationUnit cu = |[ #[classdec] ]|
```

To the user of the metalanguage it might be confusing that no actual object language syntax is involved in these assignments, bot the direct assignments `e = s` or `cu = classdec` are not type correct. Furthermore, it is natural to be able to use a list of identifiers as a list of expressions. In our generic framework, we do not support such object language specific features. Unfortunately, implementing such conversion by hand in the metaprogram takes quite some code and makes to code generator less clear.

These object language specific features could be supported in our framework by extending the type-checker of the metalanguage to support such conversions. Thus, the object language specific conversions are added to the built-in conversions of the metalanguage. The conversions could be derived automatically from a syntax definition of the object language, thus making this feature object language independent. Every nonterminal that can produce another nonterminal without any additional syntax is a candidate for an object language specific conversion.

3.7 CONCLUSION

We have extended an existing generic architecture for implementing concrete object syntax. The application of a disambiguating type-checker, that is separate from a generalized parser, is key for providing single quotation and anti-quotation operators without explicit typing. This approach differs from other approaches due to this separation of concerns, which results in object

language independence. It can still handle complex configurations such as Java embedded in Java.

We have validated our design by means of two different realistic embeddings of object languages into Java, and comparing the results to existing systems for meta programming. The instances of our framework consist of meta programming languages that use manifest typing (i.e. Java), combined with object languages that have a well-typed meta representation (e.g., Eclipse JDT Core DOM).

We explicitly do not provide heuristics to automate or infer types, such that the architecture's behavior remains fully declarative and is guaranteed to be compatible with the type system of the metaprogramming language.

ACKNOWLEDGEMENTS

At Utrecht University this research was supported by the NWO/JACQUARD project 638.001.201 TraCE: Transparent Configuration Environments, and at CWI by the Senter ICT-doorbraak project CALCE. We would like to thank Karl Trygve Kalleberg, Eelco Dolstra and the anonymous reviewers of GPCE 2005 for providing detailed feedback.