Interactive Disambiguation of Meta Programs with Concrete Object Syntax

Lennart C.L. Kats[1], Karl T. Kalleberg[2], and Eelco Visser[1]

[1]Software Engineering Research Group, Delft University of Technology, The Netherlands
l.c.l.kats@tudelft.nl, visser@acm.org
[2]KolibriFX, Oslo, Norway
karltk@kolibrifx.com

Abstract. In meta-programming with concrete object syntax, meta programs can be written using the concrete syntax of manipulated programs. Quotations of concrete syntax fragments and anti-quotations for meta-level expressions and variables are used to manipulate the abstract representation of programs. These small, isolated fragments are often ambiguous and must be explicitly disambiguated with quotation tags or types, using names from the non-terminals of the object language syntax. Discoverability of these names has been an open issue, as they depend on the (grammar) implementation and are not part of the concrete syntax of a language. Based on advances in interactive development environments, we introduce *interactive disambiguation* to address this issue, providing real-time feedback and proposing quick fixes in case of ambiguities.

1 Introduction

Meta programs analyze, transform, and generate programs. Examples include compilers, interpreters, and static analysis tools. Most frequently, meta programs operate on the abstract syntax of an object language, using a structured representation of programs rather than a textual representation of their source code. Using a structured representation ensures well-formedness, enables compositionality of transformations, and makes it easier to support type safety and hygiene. However, manipulating the abstract syntax through an API can get tedious, and larger structures are often hard to recognize.

Meta-programming with concrete object syntax [15] as a surface syntax for the abstract representation is, for a great number of situations, a best of both worlds between a textual and an abstract syntax representation: the meta program is written using the familiar concrete syntax of the object language, while at the meta level, all operations are done on a structured representation of the object program. Concrete object syntax can be syntactically checked as meta programs are compiled. This technique is now supported by many meta-programming systems [2,3,4,7,12].

A prevailing problem with embedding concrete object syntax inside a meta-language is that the syntax of the combined meta-and-object language is usually highly ambiguous when the embedding employs a single pair of quotation and anti-quotation symbols. For example, a quoted Java code fragment `|[ i = 2 ]|` can either be an assignment expression, part of a local variable declaration, or even an annotation element initializer.

Two approaches have been proposed to address ambiguity in meta programs, each with their own trade offs and limitations. Perhaps the most straightforward approach

B. Malloy, S. Staab, and M. van den Brand (Eds.): SLE 2010, LNCS 6563, pp. 327–336, 2011.

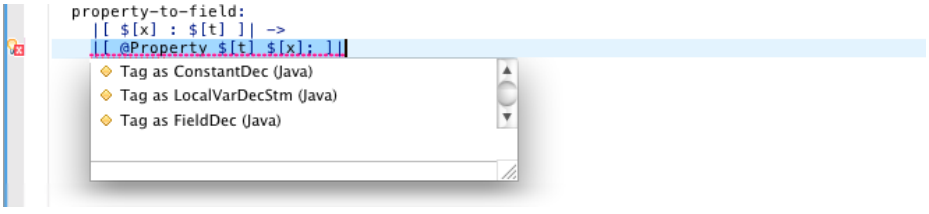

Fig. 1. Screenshot of a quick fix dropdown menu, listing three possible tags to disambiguate a Java quotation. The menu can be triggered by clicking on the error icon shown in the left margin, or by using a keyboard shortcut. Selecting a suggestion fixes the ambiguity.

is to use tagged quotation and anti-quotation symbols, e.g. writing `Expr |[ i = 2 ]|` using the tag `Expr` to indicate that the quotation contains an expression. The other approach is to use type information of the meta-programming language to attempt to select the intended interpretation of a concrete syntax quotation [5,13]. For example, for an embedding of Java in Java, a statement `Expr assign = |[ i = 2 ]|;` can be disambiguated based on the declared type `Expr`, while the quotation itself does not have to be explicitly tagged.

A pressing problem that both approaches share is a lack of *discoverability* of quotation tags and types. Meta-programmers may be intimately familiar with the concrete syntax of a language, but may not be well-grounded in the specific names of non-terminals in the syntax definition and the corresponding tag and type names. Having to know these names adds to the learning curve of meta-programming. Furthermore, as object languages evolve, or as additional object languages are added to a meta program, new ambiguities can be introduced for existing code that has not yet, or insufficiently, been explicitly disambiguated. Neither of the two approaches provides developers with adequate feedback if the developer must decide how to fix such an ambiguity.

In this paper we propose *interactive disambiguation* as a *complementary* approach to tag-based and type-based disambiguation that addresses the concern of discoverability. Our work builds on advances in interactive development environments (IDEs). Modern IDEs aid in discoverability of language features and APIs by providing features such as context-aware code completion and quick fixes. Quick fixes provide a facility to quickly fix common errors by selecting a fix from a list of suggestions. In this paper we propose to use quick fixes to present developers a list of candidate type or tag names for ambiguous concrete syntax fragments, allowing them to selectively fix problematic ambiguities and quickly discover possible fixes (illustrated in Figure 1). Our interactive disambiguation approach is *fully language independent* and does not have to be adapted for a specific meta-programming language or its type system.

2 Meta-programming with Concrete Object Syntax

In this section we recapitulate the general method for supporting concrete object syntax in meta languages and describe the problem of ambiguity. This method, as described in [15,6], is independent of the meta language used and relies on composition of the

```
action-to-java-method:
  |[ action $[Id:name] {
       $[Statement*:s*]
     }
  ]| ->
  |[ public void $[Id:name]() {
       $[Stm*:<statements-to-java> s*]
     }
  ]|
```

Fig. 2. A rewrite rule that uses concrete object syntax notation to rewrite a WebDSL action to a Java method. $s*$ is a meta variable containing a list of statements. $name$ contains an identifier.

syntax of the meta language and the object language. Based on this composition, the meta language can use *quotations* of object-level code to match and construct code fragments in the object language. In turn, quotations can include *anti-quotations* that escape from the object language code in order to include variables or expressions from the meta language. As an example, Figure 2 shows a Stratego [4] rewrite rule that uses quotations (indicated by `|[...]|`) and anti-quotations (indicated by `$[...]`) to rewrite a WebDSL [16] action definition to a Java method.

Grammar composition. Some meta-programming systems, such as Jak [3] and MetaAspectJ [8], have been specifically designed for a fixed object language. These systems use a carefully handcrafted grammar or parser for the combined meta and object language. Other systems are more flexible and can be configured for different object languages by combining the grammar for the meta and object languages, and generating a corresponding parser using a parser generator. Building flexible meta-programming systems using traditional parser generators is very difficult, because their grammars are restricted to LL or LR properties. This means that conflicts arise when the grammar of the meta language and the object language are combined [6], and these must be resolved before the meta-and-object language parser can be constructed. A further impediment to language composition found in traditional parsers is the use of a separate scanner, requiring the use of a single lexical syntax definition for the combined language.

By using a combination of SDF for syntax definition and SGLR for parsing [6,14], any object language can be embedded in any meta language [15]. SGLR supports the full class of context-free grammars, which is closed under composition. This makes it possible to compose languages simply by combining grammar modules. *Mixin grammars* can combine languages by introducing productions for quotation and anti-quotation of an object language to a meta language. Mixin grammars can be written by hand or automatically generated using a tool.

As an example of a mixin grammar, Figure 3 shows an excerpt of a grammar that embeds Java into the Stratego program transformation language. Quotation productions have the form $q_1\ osort\ q_2$ `->` $msort$ and specify that a quotation of object-language non-terminal $osort$, surrounded by (sequences of) symbols q_1 and q_2, can be used in place of meta-language non-terminal $msort$. We sometimes refer to q_1 and q_2 collectively as the *quoting symbols*. In most of our examples, q_1 is `|[` and q_2 is `]|`, or a variation thereof.

```
module Stratego-Java
imports Stratego Java[1]
exports context-free syntax
% Quotations %
  "|[" ClassDec "]|" -> Term {cons("ToMetaExpr")}
  "|[" BlockStm "]|" -> Term {cons("ToMetaExpr")}
  "|[" CompUnit "]|" -> Term {cons("ToMetaExpr")}
% Anti-quotations %
  "$[" Term "]" -> ClassDec {cons("FromMetaExpr")}
  "$[" Term "]" -> BlockStm {cons("FromMetaExpr")}
  "$[" Term "]" -> CompUnit {cons("FromMetaExpr")}
```

Fig. 3. A mixin grammar for embedding object language Java into host language Stratego. For selected Java non-terminal, the mixin defines productions for quoting and anti-quoting. `ClassDec`, `BlockStm`, `CompUnit` are defined by the `Java` grammar.

```
     "CompUnit" "|[" BlockStm "]|" -> Term {cons("ToMetaExprTagged1")}
"Java:CompUnit" "|[" BlockStm "]|" -> Term {cons("ToMetaExprTagged2")}
"$[" "BlockStm" ":" Term "]" -> BlockStm   {cons("FromMetaExprTagged")}
```

Fig. 4. Productions with tagged quoting symbols

Conversely, anti-quotation productions have the form $q_1\ msort\ q_2$ `->` $osort$. They specify that an anti-quotation of meta-language non-terminal $msort$, using quoting symbols q_1 and q_2, can be used in place of object-language non-terminal $osort$.

In our example we combine a single meta language with a single object language. It is also possible to add additional object languages or embeddings and extensions inside object languages. Using nestable quotations and anti-quotations, meta and object language expressions can be arbitrarily nested.

Ambiguity. As the meta language and object language are combined, *ambiguities* can arise in quotations and anti-quotations. Quotations and anti-quotations are ambiguous if they can be parsed in more than one way, leading to multiple possible abstract syntax representations. Ambiguities can also occur if the same quoting symbols are used for (anti-)quotation of multiple non-terminals. Such ambiguities can be avoided by using quoting symbol tags that indicate the kind of non-terminal or by using type information from the meta language [5,13]. Both approaches use names based on the non-terminals in a syntax definition for the object language. Without loss of generality, we focus on a combination of tag-based disambiguation with interactive disambiguation in this paper.

As an example of tagged quoting symbols, Figure 4 shows tagged productions tag (anti-)quotations for the `CompUnit` non-terminal. We indicate the kind of tag in the constructor of these productions. For untagged quotation productions we use `ToMetaExpr`, for productions with a non-terminal name we use `ToMetaExprTagged1` and for productions that also include a language prefix we use `ToMetaExprTagged2`. The last

[1] This example uses plain imports to combine the meta and object languages (Java and Stratego). To avoid name clashes between non-terminals of the two grammars, actual mixin grammars use parametrized imports, so that all symbols are postfixed to make them uniquely named.

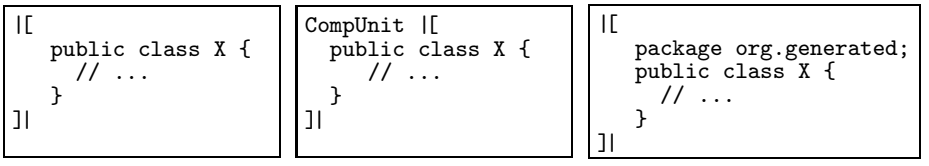

Fig. 5. Quotations of a Java compilation unit. From left to right: an ambiguous quotation, a quotation that is disambiguated by tagging, and a quotation that is already unambiguous without tagging.

category enables distinction between non-terminals with the same name that are defined in different object languages. For tagged anti-quotations we distinguish `FromMetaExpr` and `FromMetaExprTagged`.

Figure 5 shows an illustration of both untagged and tagged quotations. The quotation on the left is ambiguous, as it can represent a single class, a class declaration statement, or a compilation unit, each represented differently in the abstract syntax. The quotation in the middle makes it *explicit* that the intended non-terminal is a compilation unit, resolving the ambiguity. The quotation on the right is already unambiguous, because only complete Java compilation units can include a package declaration, and does not have to be explicitly disambiguated. Similar to quotations, anti-quotations can be ambiguous if they can represent multiple possible non-terminals within the context of a quotation.

3 Interactive Disambiguation of Concrete Object Syntax

In this section we describe how ambiguities in concrete object syntax can be interactively resolved by analyzing ambiguities and providing quick fix suggestions. We describe different classes of ambiguities and give an algorithm for automatically determining disambiguation suggestions for a given parse forest and grammar.

3.1 Classes of Ambiguities

At the grammar level, there are a number of different classes of ambiguities. In this paper we focus on ambiguities in quotations and anti-quotations. These ambiguities are inherent to the use of mixin grammars, as languages are woven together and fragments must be parsed with limited syntactic context. Disambiguation with tags or types can resolve these ambiguities. Other forms of ambiguities can be caused by the meta or object language, such as with the C language that notoriously overloads the $*$ operator for multiplication and pointer dereference. Such ambiguities must be retained if they are part of the object language design, otherwise they should be resolved at the grammar level. Ambiguities can also arise by the combination of the two languages if the syntax between the meta and object language overlap. These cannot always be resolved by type-based disambiguation [13], but can only be avoided by carefully selecting sensible quoting symbols in such a way that they do not overlap with the meta language and object language. Ideally, the symbols are chosen to be aesthetically pleasing characters or character combinations that never occur in either the object or meta language.

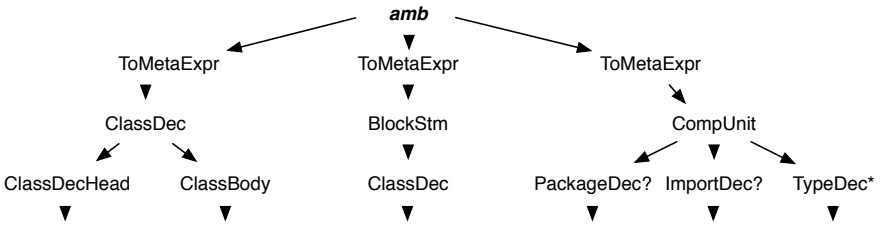

Fig. 6. The parse forest for the quotation `|[ public class X {} ]|`

Quotations are ambiguous when they can be parsed in more than one way, according to one or more object languages. To illustrate interactive disambiguation suggestions for quotations, consider again the untagged quotation in the left-hand side of Figure 5. Recall that this fragment could represent a single class, a class declaration statement, or a compilation unit. Using a generalized parser such as SGLR, the parser constructs a parse forest that branches at the point such an ambiguity, containing all possible subtrees for the ambiguous expression. Figure 6 illustrates the parse forest for our example, with at the top a special "amb" tree node that has the three possible interpretations as its children. The gist of our technique is to analyze the different possible parse trees, and have the developer select which alternative they intended.

In the mixin grammar for the embedded Java language (shown in Figure 3), there are three untagged productions that produce the three interpretations of our example. The tagged productions of Figure 4 parse the same object language non-terminal, but include distinguishing tags. These tags can be used to disambiguate the example: when one of the tags `ClassDec`, `BlockStm`, or `CompUnit` is added, there is only one possible interpretation of the quotation. By providing quick fix suggestions that automatically insert one of these three tags, meta-programmers can consider the three options and decide which is the interpretation they intended. In the event that the fragment could also be parsed using a different object language that happens to use the same tag names, the prefixed tags such as `Java:CompUnit` are proposed instead.

Anti-quotations can be disambiguated much like quotations. However, because they always occur in the context of a quotation, there is no need for language-prefixed quoting tags. For anti-quotations we also distinguish *local ambiguity*, where a single anti-quotation can be parsed in multiple ways, and *non-local ambiguity*, where a larger area of the quotation can be parsed in multiple ways. Non-local ambiguities arise as anti-quotations productions typically reduce to multiple possible non-terminals, whereas quotation productions typically reduce to only one, such as `Term` in Figure 3.

Figure 7 (left) shows a local ambiguity. The anti-quotation `$[`x`]` may be interpreted as an identifier or as the signature of the quoted class. The remainder of the quotation is unambiguous, making it trivial to identify the cause of the ambiguity in the parse forest.

Figure 7 (right) shows a non-local ambiguity. For this example, the entire body of the quotation can be interpreted in multiple ways: it can be either a class y with modifier x, or a package/import/type declaration x followed by a class y. For non-local ambiguities it is harder to identify the cause of the ambiguity, as the quotation expressions are no longer a direct subtree of the "amb" node as they are in Figure 6.

```
CompUnit |[
  class $[x] {
    // ...
  }
]|
```

```
CompUnit |[
  $[x] class Y {
    // ...
  }
]|
```

Fig. 7. Example of a local ambiguity (left) and a non-local ambiguity (right). In the first only the anti-quotation `$[x]` is ambiguous, in the other the entire contents of the quotation is ambiguous.

3.2 Automatic Disambiguation Suggestions

In this subsection we describe an algorithm to automatically collect disambiguation suggestions. We implemented the algorithm using Stratego and published the implementation and source code online at [1]. A prototype currently integrates into the Spoofax language workbench [10].

Figure 8 shows pseudocode for the disambiguation suggestions algorithm. At the top, the `CollectSuggestionsTop` function is the main entry point, which gets the parse forest and grammar as its input and returns a set of disambiguation suggestions as its output. For each outermost ambiguous subtree amb, it uses the `CollectSuggestions` function to find local disambiguation suggestions.

The `CollectSuggestions` function produces a set of disambiguation suggestions by inspecting each subtree of the amb tree node (line 2). For each branch, it searches for the outermost meta-expressions that are not yet completely tagged (line 3). For each meta-expression it determines the production $prod$ that was used to parse it (line 4), and its left-hand and right-hand side non-terminals (line 5, 6). For SGLR parse trees, the production is encoded directly in the tree node, allowing it to be easily extracted. Only meta-expressions that are a direct child of amb (local ambiguities) and meta-expression subtrees that do not have any tag (non-local ambiguities) are considered for suggestions (line 7).[2] For the selected meta-expressions, a set of possible disambiguation suggestions is collected (line 8). These suggestions take the form of tagged meta-expression productions (line 9) that contain the same left-hand and right-hand side non-terminals as the production $prod$ (line 10). Of course, we only include quotation productions if the current expression is a quotation, and anti-quotation productions if it is an anti-quotation (line 11). After all corresponding suggestions are collected, the complete set is filtered using the `FilterAmbiguousSuggestions` function (line 13).

The `FilterAmbiguousSuggestions` function filters out any suggestions that are ambiguous with respect to each other. This is useful if two object languages both match a meta-expression *and* they use the same quotation tag `X`. In those cases, inserting the tag `X` would not resolve the ambiguity, and a tag with a language prefix of the form `Lang:X` must be proposed instead. For suggestions with quoting symbols q_1,q_2 (line 3, 4), the function only returns those for which there is no other suggestion with the same quoting symbols (line 5, 6).

[2] A special case is the `ToMetaExprTagged1` constructor, used for tagged quotations without a language prefix. Suggestions are only provided for *local* ambiguities with this constructor.

```
COLLECTSUGGESTIONSTOP(tree, grammar)
1  results ← {}
2  foreach outermost subtree amb in tree where amb has the form amb(...)
3     results ← results ∪ COLLECTSUGGESTIONS(amb, grammar)
4  return results

COLLECTSUGGESTIONS(amb, grammar)
 1  results ← {}
 2  foreach child subtree branch in amb
 3     foreach outermost subtree expr in amb where ISMETAEXPRTAGGABLE(expr)
 4           prod ← the production for expr
 5           lsort ← the non-terminal at left-hand side of prod
 6           rsort ← the non-terminal at right-hand side of prod
 7           if expr = branch ∨¬ISMETAEXPRTAGGED(prod) then
 8              results ← results ∪ { (expr,prod′) | prod′ ∈ productions of grammar
 9                                                 ∧ ISMETAEXPRTAGGED(prod′)
10                                                 ∧ prod′ has the form (q1 lsort q2 -> rsort)
11                                                 ∧ prod′ and prod have the same construc-
12                                                   tor prefix To or From }
13  return FILTERAMBIGUOUSSUGGESTIONS(results)

ISMETAEXPRTAGGABLE(t)
1  if t has FromMetaExpr, ToMetaExpr, or ToMetaExprTagged1 constructor
2  then return true
3  else return false

ISMETAEXPRTAGGED(p)
1  if p has FromMetaExprTagged, ToMetaExprTagged1 or ToMetaExprTagged2 constructor
2  then return true
3  else return false

FILTERAMBIGUOUSSUGGESTIONS(suggestions)
1  return { (prod,expr) | (prod,expr) ∈ suggestions
2                       ∧ prod has the form (q1 lsort q2 -> rsort)
3                       ∧¬∃ (expr′,prod′) ∈ suggestions:
4                             prod′ has the form (q1 lsort′ q2 -> rsort′) }
```

Fig. 8. Pseudo-code for collecting suggested quotation symbols

3.3 Presentation of Suggestions

Interactive disambiguation is based on the notion of *quick fixes*, small program transformations that can be triggered by the developer in case of a code inconsistency or code smell. Quick fixes are non-intrusive: as developers write their program, errors or warnings are marked inline, but it is up to the developer to decide when and if to address the problems. For interactive disambiguation, quick fixes allow meta-programmers to write concrete syntax for expressions first, allowing the parser to decide whether or not it is ambiguous, proposing appropriate quick fixes as necessary.

The `CollectSuggestionsTop` function is executed each time the result of parsing the meta program is ambiguous. The quickfix menu is populated with the results, and any ambiguity can be addressed by adding the tag name or by inserting the type into the context of the quotation. In order to avoid spurious suggestions for multiple ambiguities, we only provides suggestions for the *outermost* expressions (`Collect-Suggestions`, line 8), allowing meta-programmers to incrementally fix any remaining ambiguities.

While we emphasize interactivity, it should be noted that the technique does not necessarily require an IDE. Quotation alternatives can also be displayed as part of the build process and used with generic text editors that may not interactively parse and analyze the meta language source code.

In our implementation we cache operations such as collecting productions from the grammar for efficiency, while in the algorithm described here we abstract from these optimizations. Experience with the prototype tells us that the performance overhead of the suggestions algorithm is very low, as it only does a depth-first traversal of each ambiguity and a few hash table lookups.

4 Discussion and Conclusions

In this paper we combined interactive and tag-based disambiguation to reduce quotation noise in meta-programs with concrete syntax. Developers only need to quote where absolutely necessary, and are interactively helped to introduce appropriate quotation symbols where required.

Interactive disambiguation can also be combined with type-based disambiguation, assisting in cases where type-based disambiguation is inadequate, as multiple type-based interpretations are type correct. These cases particularly arise when combining the technique with type inference, as seen with Meta-AspectJ [8], or when forgoing quoting symbols that distinguish between the meta and the object language, as observed by Vinju [13]. Both works propose heuristics as a solution in these cases. Interactive disambiguation can let the programmer interactively, and thus more predictably, resolve such ambiguities statically. Alternatively it can assist when programs are not yet type consistent, providing suggestions for inserting type declarations or type casts. Stratego is largely untyped, ruling out type-based disambiguation for our present prototype. A typed variant of Stratego [11] might be a suitable testbed for experiments combining interactive disambiguation and type inference. On the dynamic side, we have had promising experimental results using *runtime disambiguation*, where the decision of the correct interpretation of a meta-expression is delayed until run-time, when the actual values of meta-level expressions are known. Based on a static analysis of the meta-program, it is possible to determine which quotations can safely be disambiguated at runtime.

We performed a preliminary evaluation of our approach using existing source files that embed Java in Stratego, from the Dryad Java compiler [9]. The sources use a total of 55 concrete syntax quotations of a wide variety of different Java language constructs. Most are small quotations, but a few contain complete compilation units, used for compilation and for unit testing. We stripped all existing disambiguation tags from the sources, and by following the interactive disambiguation suggestions, were able to

successfully disambiguate the files. We then introduced WebDSL [16] as an additional object language as a form of an evolution scenario. This introduced new ambiguities, as some expressions such as `Expr |[ $[Expr:`x`] == $[Expr:`y`] ]|` would be a valid quotation for either language. Again, applying the quick fixes helped the transition and resolved the ambiguities by introducing language-prefixed tags.

We have found interactive disambiguation to be a practically useful technique, complementary approach to both tag-based [6], and type-based disambiguation [5,13], and independent of the meta and object language and their type system.

Acknowledgements. This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

1. The interactive disambiguation project (2010), `http://strategoxt.org/Spoofax/InteractiveDisambiguation`
2. Arnoldus, J., Bijpost, J., van den Brand, M.: Repleo: a syntax-safe template engine. In: GPCE, pp. 25–32 (2007)
3. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for Implementing Domain-Specific Languages. In: Conference on Software Reuse, p. 143 (1998)
4. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. SCP 72(1-2), 52–70 (2008)
5. Bravenboer, M., Vermaas, R., Vinju, J.J., Visser, E.: Generalized type-based disambiguation of meta programs with concrete object syntax. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 157–172. Springer, Heidelberg (2005)
6. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: OOPSLA, pp. 365–383 (2004)
7. Cordy, J.R., Halpern-Hamu, C.D., Promislow, E.: TXL: A rapid prototyping system for programming language dialects. Comp. Lang., Syst. & Struct. 16(1), 97–107 (1991)
8. Huang, S.S., Zook, D., Smaragdakis, Y.: Domain-specific languages and program generation with Meta-AspectJ. Transactions on Software Engineering Methodology 18(2) (2008)
9. Kats, L.C.L., Bravenboer, M., Visser, E.: Mixing source and bytecode: a case for compilation by normalization. In: OOPSLA, pp. 91–108 (2008)
10. Kats, L.C.L., Visser, E.: The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In: OOPSLA, pp. 444–463 (2010)
11. Lämmel, R.: Typed generic traversal with term rewriting strategies. Journal of Logic and Algebraic Programming 54(1), 1–64 (2003)
12. van Deursen, A., Heering, J., Klint, P. (eds.): Language Prototyping. An Algebraic Specification Approach. AMAST Series in Computing, vol. 5. World Scientific, Singapore (1996)
13. Vinju, J.J.: Type-driven automatic quotation of concrete object code in meta programs. In: Guelfi, N., Savidis, A. (eds.) RISE 2005. LNCS, vol. 3943, pp. 97–112. Springer, Heidelberg (2006)
14. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
15. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Blum, A., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)
16. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)