C/C++ Disambiguation Using Attribute Grammars

Valentin David Akim Demaille Renaud Durlin Olivier Gournet

Epita Research and Development Laboratory (LRDE)
http://transformers.lrde.epita.fr <transformers@lrde.epita.fr>

Abstract

We propose a novel approach to semantics driven disambiguation based on Attribute Grammars (AGs). AGs share the same modularity model as its host grammar language, here Syntax Definition Formalism (SDF), what makes them particularly attractive for working on unstable grammars, or grammar extensions. The framework we propose is effective, since a full ISO-C99 disambigution chain already works, and the core of the hardest ambiguities of C++ is solved. This requires specific techniques, and some extensions to the stock AG model.

1 Introduction

1.1 The Transformers Project

In order to implement a fast and generic image processing library, the EPITA Research and Development Laboratory (LRDE) chose a language that supports several paradigms and never sacrifices speed for features. Back at the end of the 90's, C++ was the only reasonable answer, unchallenged as of today thanks to its incredibly powerful `template` mechanism allowing meta-programs (i.e., compile-time programs). Unfortunately, inheriting from C, a language designed in the early 70's, C++ features a poorly designed syntax and baroque semantics. This is troublesome both to compiler writers (most C++ crunching programs are not compliant and fail to capture the whole language) and to C++ writers. Indeed, `template` intensive programming is error-prone and often incomprehensible. Therefore we considered building a new language with better meta-programming support, a daunting task in itself, and rather focused on means to add syntactic sugar to C++. This prompted for the inception of the Transformers project, aiming at providing a C++ transformation framework, i.e., a modular C++ front-end (and "back-end": a pretty-printer), so as to enable its users to develop transformations.

1.2 C++

The C++ programming language is an object oriented extension to C. It is large and complex. It inherits from the old and well known ambiguities of C. For instance, parsing `a * b;` depends on the context: if `a` is a variable name, it denotes a product, and if `a` is a type name, it denotes the declaration of `b`. Many similar ambiguities exist, and their common root is the spirit in which the C++ grammar appears to be written: it is tailored for compilers which parser provide feed-back to the scanner to escape from context-free grammar by being able to distinguish different identifier types (variable name, type name, etc.). Clearly, the C++ grammar was written with implementations in mind, not with the language itself.

C++ also adds ambiguities of its own, even with known identifier kinds. For instance, let `T` be a type, depending on the context `T(a);` denotes either the declaration of the variable `a`, or a call to the constructor `T::T(a)`. According to the standard, the latter is the correct interpretation, unless it cannot be correct in the context. The `template` keyword comes with its set of specific issues, a single of which is be presented here, Fig. 1.

1.3 Semantics Driven Disambiguation

Because we aim at using Stratego/XT to write transformations (such as syntactic sugar) and in order to enjoy C++ concrete syntax in Stratego code, we chose to use regular "Stratego/XT

```
1 template <int>
2 struct A { typedef int t; };     // A<i>::t is a type (by default).
3
4 template <>
5 struct A<0> { static int t; };  // A<0>::t is a value.
6
7 int v(A<0>::t);  // Defines a variable since A<0>::t is a value.
8 int f(A<1>::t);  // Declares a function since A<1>::t is a type.
```

The last two lines show the two possibilities for a single ambiguity: `int a(b);` may denote the definition of the variable `a` if `b` is a value (`int zero(0);`), or the declaration of a function if it's a type (`int abs(int);`). To disambiguate, the class template `A` must be instantiated with its actual parameter, since there is no requirement that a symbol (`t`) defined by a class template (`A`) has a constant kind (value or type).

Figure 1: Ambiguities on `A<?>::t` need `A` instantiations

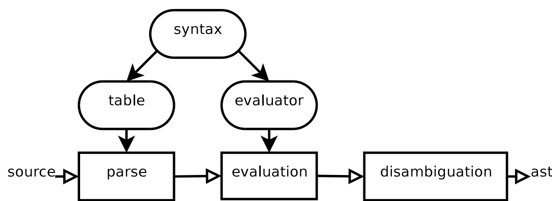

Figure 2: Disambiguation process

parsing techniques" (Klint and Visser, 1994): SDF to define a context free super set of the C++ syntax, Scanner-less Generalized LR (SGLR) to parse possibly ambiguous C++ programs and to return an AsFix parse forest, and then a set of disambiguation filters. However our approach relies on AGs to disambiguate: computations similar to type checking are performed on the attributes, revealing the invalid alternatives of ambiguities. These branches are flagged, and pruned in a latter stage. We use a homegrown extension to SDF with attribute rules embedded (Borghi et al., 2005). Fig. 2 presents the parsing chain, from source code to (unique) Abstract Syntax Tree (AST).

2 Disambiguating with AGs

The formalism of AGs is well defined, and well covered in the literature. Yet little attention was devoted to using *ambiguous* AGs: how can information flow when there is uncertainty. As usual, C++ added on top of this its own issues, requiring an extension to AGs.

2.1 Ambiguous synthesis

In attribute grammars, evaluation consists in information flowing across nodes of the Parse Tree (PT), either upwards (*synthesized* attributes), or downwards (*inherited* attributes). Extension of inherited attributes flow in ambiguity nodes is straightforward: the node simply forwards the unique information flowing downwards to its children. Synthesized attributes are a problem: each children contributes a possibly different value. In our model, if the values are equal, this value is assigned to the ambiguity node, but if they are different, an error is raised and the evaluation stops. None of our current applications required a more subtle policy. In the future, we might consider ambiguity support for attributes, similar to the `amb` node in AsFix. A formal treatment of our extension of AGs to ambiguity remains to be done.

2.2 Template instances

C++ `templates` challenge the compiler writers: they are the number one reason for lack of compliance with the C++ standard. There is no exception for AGs: special mechanisms are required to cope with this part of the language.

One issue with parsing (and disambiguating) (class or function) templates is that their context is incomplete. Consider for instance the Fig. 3: what is the kind of the symbol `T::t`? It turns out that in this case the standard made provisions to turn this context sensitive problem into context-free thanks to the `typename` keyword.

```
// Is T::t a type, or a value?
template <typename T>
void g() { int f(typename T::t); }

// Is T::t a type, or a value?
template <typename T>
void h() { int f(T::t); }
```

In the definition of `g`, using the keyword `typename`, the programmer declares any actual parameter will define `t` as a type. Conversely, for `h`, the absence of `typename` *requires* the user to provide actual parameters that define `t` as a value. In this specific case (extracting a symbol from a template parameter) the C++ standard mandates explicit disambiguation.

Figure 3: The mandatory `typename` disambiguation keyword

```
// A<I>::t is A<I-1>::t.
template <int I>
struct A : public A <I-1> {};

// A<0>::t is a value.
template <>
struct A<0> { static int t; };

// A<14>::t is a type.
template <>
struct A<14> { typedef int t; };
```

Figure 4: Recursive template instantiations

Therefore template definitions are easily handled. However, template *uses* are much more delicate: they require that templates be (partially) instantiated. Consider for instance the Fig. 1: disambiguating lines 7 and 8 require the instantiation of `A` with its actual parameters. To this end, AGs are troublesome. One simple solution involves two passes: the first pass gathers the actual template parameters (0 and 1 in our example), and the second pass instantiates the template definition with them, finally providing the dependent information (the kind of `A<?>::t`) to disambiguate (`v` is a variable, and `f` a function). Unfortunately because template instantiations can trigger arbitrarily many other template instantiations (Fig. 4) this is not possible.

Therefore *the template must be carried from its definition to its uses* together with its set of attribute rules, since they have to be evaluated "on site". This requires a dramatic extension to AGs: part of a PT is a possible attribute value, and a means to fire the evaluation of attributes is needed.

As a consequence, the template definitions are no longer evaluated where they are defined, but where they are used. This is insufficient though: errors in the template definitions must also be caught to comply with standard C++.

There are two options to evaluate template definitions. The simplest solution consists in instantiating the template with fake parameters. Alternatively, one could exploit information gathered from the uses of a template to disambiguate its body. Consider Fig. 3, line 5: any use of this template provides an actual value for `T`, hence an explicit definition for `T::t`. This framework would relieve the user from having to use the `typename` keyword to disambiguate by hand template uses.

3 Discussion

3.1 Results

We developed a complete AG for the ISO-C99 language. Our grammar strictly conforms to the standard in about 340 rules (half of which are lexical) and 90 attributes. Contrary to C++, it has few and easy-to-resolve ambiguities, making it a realistic test bed case. Its development represented about 1-2 month-man. The sheer size of this grammar prompted for the addition of AG syntax extension in order to minimize code duplication. In the long run, when a fully blown redesign of AG in SDF is completed, extensions such as featured by Utrecht University Attribute Grammar System (UU-AG) (Swierstra et al., 2003) will be implemented. Performances are, as expected from a naive implementation, poor: disambiguating `stdio.h` takes 75s on a 3GHz microprocessor.

The extension of this grammar to support GNU C is straightforward, and requires no modification to the baseline grammar. C++ is a much more challenging grammar to tackle (560 rules and more and much harder ambiguities). As a proof of concept, a mini C++ grammar was developed to include all the difficult aspects of C++ disambiguation (virtually everything related to templates). Its extension into a disam-

biguating AG is completed, which supports our claim that AG can disambiguate C++.

3.2 Others solutions

We propose the use of AGs to perform semantics driven disambiguation, but other techniques have been applied with success.

van den Brand et al. (2003) use **ASF+SDF** to disambiguate ambiguous parse trees. Nevertheless, according to our initial experiments, this approach is delicate to extend to a fully blown language. It remains yet to be proved that templates can be properly handled.

Our first experiments using **Stratego** did not use the recently introduced dynamic rules, and therefore the code was entangled with scopes and tables processing. In addition, because a primary motivation for the inception of the Transformers project is to ease the implementation of C++ grammar extensions, we looked for seamless modularity. Embedding the disambiguation specifications in the grammar provides modularity for free.

3.3 Further works

ASF+SDF, Stratego, and AGs provide three different means to write elegantly and concisely disambiguation filters. Because of template instantiation, C++ challenges these techniques, and a thorough comparison between the three paradigms is underway (Vasseur, 2004). Once completed, this comparison will address a small subset of C++ containing the following features: modularity by modeling C++ as an extension of C, templates to mandate instantiations (comparable to extending the PT during its traversal), namespaces to introduce named scopes, and context sensitivity (by introducing two kinds in C, `union` and `typedef`, and a third for C++, `class`).

A formalization of our extensions to stock AGs remains to be done. Yet our model is still slightly changing, tailored to ease the implementation of disambiguation filters. For instance, it is considered to allow the evaluator to prune incorrect branches, sort of a `cut`, instead of merely flagging them as incorrect. Early experiment show a small speedup, but significant simplifications of attribute rules.

Finally, to implement actual transformations we wish to use C++ concrete syntax in Stratego, what prevents C++'s intrinsic ambiguity. We are toying with the idea of using AGs to disambiguate embedded languages. A successful early experiment allows the programmer to disambiguate C in Stratego by hand as follows.

```
mytest = ?|Expression[ (i0) (i1) ]|
           with i0 => "typedef",
                i1 => "variable"
```

4 Conclusion

We demonstrated how (ambiguous) AGs can be used to perform semantics driven disambiguation. The disambiguation of difficult languages demonstrate the validity of the approach: C99 is fully covered, and the most delicate parts of C++ have been solved. It is ongoing work to address the full language. AGs are modular and extendable, which we shall use to implement grammar extensions, and explore embedded languages disambiguation.

References

Borghi, A., David, V., Demaille, A., and Gournet, O. (2005). Implementing attributes in SDF. Submitted to Stratego Users Day 2005.

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

Swierstra, S. D., Baars, A., and Löh, A. (2003). The UU-AG attribute grammar system. `http://www.cs.uu.nl/groups/ST`.

van den Brand, M., Klusener, S., Moonen, L., and Vinju, J. J. (2003). Generalized parsing and term rewriting: Semantics driven disambiguation. volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Vasseur, C. (2004). Semantics driven disambiguation: a comparison of different approaches. Technical report, LRDE. `http://publis.lrde.epita.fr/20041201-Seminar-Vasseur-Disambiguation-Report`.