C-Transformers
A Framework to Write C Program Transformations

Alexandre Borghi, Valentin David, Akim Demaille
First.Last@lrde.epita.fr
http://transformers.lrde.epita.fr
EPITA Research and Development Laboratory (LRDE)

December 11, 2006

Abstract

Program transformation techniques have reached a maturity level that allows processing high-level language sources in new ways. Not only do they revolutionize the implementation of compilers and interpreters, but with modularity as a design philosophy, they also permit the seamless extension of the syntax and semantics of existing programming languages. The C-Transformers project provides a transformation environment for C, a language that proves to be hard to transform. We demonstrate the effectiveness of C-Transformers by extending C's instructions and control flow to support Design by Contract. C-Transformers is developed by members of the LRDE: EPITA undergraduate students.

1 Introduction

New trends in programming languages set a new challenge to the researcher: productivity. One trend focuses on providing the programmers with more productive languages, to this end *program transformations* techniques are extremely powerful. To implement these transformations, *language-specific frameworks* are needed. To compose these frameworks, *language agnostic tools* are needed that can be used to quickly address a particular issue in whatever language.

Program transformations encompass virtually every type of program processing. **Data extraction** from sources allow to perform code metrics (number of lines, statements, function arguments, maintainability etc.), to generate documentation. **Software renovation** consists in improving existing code, possibly very large and ancient programs. Even when a thorough design was made, when experienced programmers are involved, when high level development tools are used etc. refactoring is needed to keep a program healthy: maintainable. Programmers are used to refactor by hand, but today much can be mechanized, including the use of advanced constructs such as Design Patterns [34]. The development of such refactoring tools for Integrated Development Environments is an active topic of research. **Optimization** is another active field, including domain-specific optimizations [3] and standard optimizations, possibly very high level ones such as partial evaluation [26]. **Translations**, such as compilation, typically make use of program transformation techniques, with several refining steps. Most of the time transformations are written in a general language (C, C++, Caml...), but some exploit dedicated techniques (e.g., Tiger in Stratego [31], or even the Stratego Compiler itself [30]). As a specific instance of translation, **language extension** provides an existing language with additional features and write a translator that "compiles" (**assimilation** in the words of [7]) the extended language down to the host language. As a running example, this paper will focus on such an application.

Language-specific frameworks. Writing a compiler is a tremendous task, and the implementation of a transformation framework makes no exception: one needs (i) a parser to read the

input, (ii) possibly a disambiguation step, (iii) optionally a type checker, (iv) the core transformations themselves, and finally (v) a pretty printer to convert the program back to text. Every step of this framework is language-specific. The infrastructure can outweigh the transformation by far, therefore, in order to make the transformation implementation productive, language-specific frameworks are needed. In this paper we present the C-Transformers project [19], a framework enabling the seamless implementation of transformations programs in C— or C variations.

Language generic components. Vast research and many tools were developed to provide the technology needed to implement each type of component of the framework, sometimes leading to nice generalizations. It is well known for instance, that there are tools that generate parsers from grammars; it is less known is that other steps also enjoy the existence of language generic components that can easily be tailored to a specific language [16]. C-Transformers is using Stratego/XT [15] [32] as its library of language generic components. These off-the-shelf tools allow us to focus directly on C-specific issues (e.g., its disambiguation).

The C-Transformers is a free software project available on the Internet [19] developed by EPITA undergraduate students. EPITA is a French private engineering school dedicated to computer science. The Research and Development Laboratory of EPITA, LRDE, recruits amongst the best students willing to follow a more academic curriculum, possibly leading to a PhD. While members of the LRDE, they work on research topics supervised by assistant professors. V. David and A. Borghi are members of this group working actively on disambiguation under the supervision of Akim Demaille.

C-Transformers can be used to implement virtually any kind of C program transformation, but currently, because of its lack of a reversible pre-processor (Section 6) it cannot be used for software renovation.

The Olena [11, 20] and Vaucanson [18, 22] projects gave the LRDE a strong experience in the development of C++ fast and generic libraries. New C++ programming techniques were then invented [8], unfortunately resulting in somewhat obfuscated code. The Transformers project was created to address this issue, for instance by adding syntactic sugar to C++.

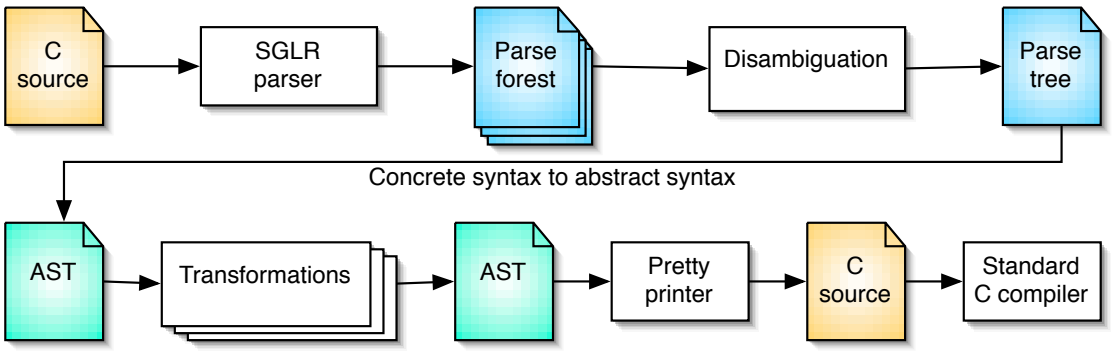
The structure of this paper is as follows. Section 2 presents similar projects. Section 3 introduces the C-Transformers chain components. In Section 4, to demonstrate its use, this C generic chain will be used to implement an extension of C, from its parsing down to its compilation to (ISO) C. Section 5 is devoted to deeper discussion on the results. Then Section 6 concludes and presents leads for future work.

2 Related Work

Program transformation draws a lot of attention these days, with several existing projects. Three of them are specially close of ours, CIL addresses ISO C, MetaBorg has the exact same goals but focuses on Java, and Proteus share goals, techniques, and target language.

CIL [24, 25] is an extremely complete and mature front end for the C language. It also includes a type checker, and a normalization of C towards a clean and simpler subset of C. This considerably eases the transformation of C programs by reducing the number of cases to handle. Nevertheless CIL does not (and cannot) provide some of the features that prompted the development of C-Transformers. (i) Transformations for CIL are more naturally expressed in Caml, the language it is written in. This does not prevent using CIL as a front-end to a Stratego program, but glue code is needed. (ii) CIL's grammar is hardwired and thus cannot be easily extended, especially not in a modular way. (iii) Using an Syntax Definition Formalism (SDF) grammar and using SDF tools is mandated to enjoy concrete syntax in Stratego. (iv) CIL's output is not "syntactically faithful", i.e., the output program is semantically equivalent but it is a deep modification of the input program, with a complete loss of layout, comments, and preprocessor directives. CIL is inadequate for code factoring.

In the words of its authors [6, 7], "**MetaBorg** provides generic technology for allowing a host language (collective) to incorporate and assimilate external domains (cultures) in order to



Given a C grammar written in SDF (Section 3.1), the SGLR parser reads the text and yields a set of parse trees: a parse forest (Section 3.2). A disambiguation step keeps a single parse tree (Section 3.3), transformed into an AST suitable for the transformation(s) (Section 3.4). Finally the AST is converted back into compiler-ready C source text, a process named pretty-printing (Section 3.5).

Figure 1: The C-Transformers chain

strengthen itself. The ease of implementing embeddings makes resistance futile". Basically the MetaBorg project has exactly the same purpose as the Transformers project, but on a different host language: while Transformers focuses on C and C++, MetaBorg started with Java. Java is a much cleaner language than C, let alone C++ for which the development of a parser and disambiguation filter proved to be a daunting task. The MetaBorg chain relies on the exact same tools except for semantics driven disambiguation, which is written directly in Stratego (described in 3.4). In [6] the power of the system is demonstrated by several extensions of Java to include Domain Specific Languages (DSLs): seamless syntax and semantics for XML, Swing, and even Java programs. Such a tour de force was made possible thanks to the modularity of the suite (Section 5). Contrary to Transformers they already have a type checker, which allows to simplify the syntax even further by taking types into account during the disambiguation.

The goals of the **Proteus** [33] are extremely similar to Transformers': building a C transformation framework that makes it possible to preserve the programming style. These projects share many tools (SDF, Scanner-less Generalized LR (SGLR), Stratego), but they differ on some aspects. We strictly adhere to the standard grammar, they tailored theirs; we have a workable solution to disambiguate C and extended C Abstract Syntax Trees (ASTs), but their paper does not mention disambiguation; we stick to Stratego with C concrete syntax, they introduce an other language, YATL, compiled into Stratego; we explicitly want to experiment grammar extensions, they focus on standard C program refactoring, finally Transformers is free software [19]. The initial development effort also differs: Transformers attacks the modular disambiguation of C (and C++) first, and Proteus first made sure they can preserve the programming style — not only comments and layout, but also preprocessor directives.

3 The Transformation Chain

In this section we present the C-Transformers framework, component by component. Figure 1 schematizes the whole process.

```
module Declarators
imports ConstantExpressions TypeQualifiers ParameterDeclarations
exports
  sorts Declarator DirectDeclarator Pointer PointerSeq
  context-free syntax
    PointerSeq? DirectDeclarator              → Declarator
    Identifier                                → DirectDeclarator
    "(" Declarator ")"                        → DirectDeclarator
    DirectDeclarator "[" TypeQualifierList? AssignmentExpression? "]"
                                              → DirectDeclarator
    DirectDeclarator "[" "static" TypeQualifierList? AssignmentExpression "]"
                                              → DirectDeclarator
    DirectDeclarator "[" TypeQualifierList "static" AssignmentExpression "]"
                                              → DirectDeclarator
    DirectDeclarator "[" TypeQualifierList? "*" "]"
                                              → DirectDeclarator
    DirectDeclarator "(" ParameterTypeList ")" → DirectDeclarator
    DirectDeclarator "(" IdentifierList? ")"   → DirectDeclarator
    Pointer+                                  → PointerSeq
    "*" TypeQualifierList?                    → Pointer
```

Contrary to the (E)BNF, production rules are rather oriented as "reduction rules". This excerpt of a C grammar module focuses on function "declarators", i.e., signatures that are used both to declare and to define functions. The interface of the module specifies that it provides the symbols `Declarator`,..., `PointerSeq`, and requires from other modules the symbols `ConstantExpressions` etc.

Figure 2: SDF excerpt of the C grammar

3.1 SDF Grammars

The Stratego/XT tool set uses SDF [28] as its backbone: the tools are parametrized by an SDF grammar specifying the language at hand. In other words, grammars are contracts [16], therefore it must be carefully crafted. This grammar syntax is modular, which improves maintainability and extensibility, by splitting the grammar in several modules. It is also extensible: additional information can be packed in the grammar via annotations.

Roughly the grammar can be written following two guide lines. It can be designed to be simple, making full use of SDF capabilities to handle precedence, etc. This is an attractive way since it results in rather short and elegant ASTs. We made "the" other choice: to stick rigorously to the grammar defined in the ISO C standard [13] in order to guarantee our strict compliance with the standard, and to provide an environment of choice to experiment extensions to the standard (which is precisely the theme covered in Section 4). As a result our ASTs are somewhat more convoluted — for instance the AST for a simple `return 42;` is 26 nodes deep.

The C grammar counts 126 symbols and 356 rules. To ease the maintenance, the grammar is split into 53 small, manageable, sub-grammars. The boundaries of these sub-grammars were chosen to address coherent, atomic, related issues; they are finer than those of the standard which breaks the grammar in only 4 parts [13, Annex A].

Figure 2 demonstrates some of SDF features.

The running example of Section 4 will extend `Declarator` to support an additional form of function declaration (Section 4.2).

3.2 SGLR and Parse Forests

A technology supporting ambiguity and yielding parse forests is needed. Amongst available techniques Generalized LR (GLR) is most attractive [27]. Not only does a generalized parser relieve us from obfuscating the grammar to cope with the limitations of the parsing technology

```c
#include <stdio.h>
int main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

Figure 3: Hello world, the famous "first" C program

— such as the infamous shift/reduce or reduce/reduce conflicts — it is actually indispensable to have the necessary level of modularity (see Section 5).

Stratego/XT uses the state-of-the-art SGLR [29] parser. It provides all the required features, modularity and ambiguity support, and produces parse forests efficiently encoded thanks to the ATerm library [4], sporting small memory footprint and maximal sharing of common subtrees. Actually, it is an "ambiguous parse tree" that is built, with `amb` nodes grouping alternative (sub-) parses in a more useful way than genuine parse forests.

As another consequence from having chosen to use the ISO C99 standard grammar verbatim, we inherit its syntactic ambiguities, some of them being "gratuitous", others requiring more powerful context sensitive disambiguation techniques. The most typical context sensitivity of C is its dependency on identifier types. For instance depending on the kind of entity the identifier `a` was associated with, `(a) * (b)` might cast `*b` to type `a`, or, multiply `a` by `b` — symbols denoting unary and binary operators exhibit the same ambiguity, e.g., `-`, `+` and `&`.

Consider Figure 3 as a running example. This program (text) is the input provided to SGLR, which will result in a parse forest. Besides the numerous ambiguities within the `stdio.h` file, there is one in the `main` part: `printf`, according to the ISO C99 standard grammar, can be either the name of an enumeration constant, or an identifier (i.e., for a variable, type, or function). Figure 4 precisely shows the two production rules in competition.

3.3 Disambiguation

In the tradition of Yacc, context-sensitive ambiguities are addressed by an elaborate cooperation between the parser and the scanner, maintaining a common table of symbols — was the identifier `a` declared to denote a type, or a variable? This results in a deterministic parsing: at most one parse tree is found. On the contrary, in the SGLR approach the ambiguous AST is traversed to gather context-sensitive information used to prune invalid parses (an approach called "semantics driven disambiguation" by [5]).

Because the C-Transformers projects aims at modularity and extensibility, we wanted to (i) embed the disambiguation filters in the SDF grammar (thus enjoying modularity for free), (ii) be declarative, and (iii) relieve as much as possible the programmer from specifying the order and types of tree traversals. Attribute Grammars (AGs) fit very well these constraints.

Attribute grammars [17] is a formalism that supports syntax directed semantic analysis: each (grammar) rule is decorated with a set of equations that relate a node's **attributes** with those of its neighbors. AGs allow to focus on local aspects, leaving the global evaluation order aside, under the responsibility of a generic engine. Although AGs cannot modify the trees, their use for disambiguation is straightforward. Attributes convey information, e.g., a symbol table. Conflicting branches of the parse forest are flagged, and a (language generic) filter is run afterward on the parse forest, pruning inconsistent alternatives.

Since no AG engine existed for SDF, we developed one. Attribute rules are embedded in the SDF grammar as additional annotations.

Figure 4 demonstrates the use of AG to disambiguate C. The performances of the system are very satisfying (see Section 5): disambiguation is negligible compared to the whole parsing (including conversion into AST).

```
Identifier → PrimaryExpression
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  (Variable <+ Function)
       )}

Identifier → EnumerationConstant
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  Enum
       )}

Identifier → TypedefName
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  Typedef
       )}
```

This example focuses on an ambiguity of C: an identifier `foo` might denote a variable, a function, a value of an enumeration type, or a type name. When traversing a node of the first kind, make sure that the `Identifier` was declared to be a variable or a function. If not, mark the node `root`'s attribute `ok` to be failed. This will prune this (incorrect) alternative from the associated ambiguity node. The other cases are similar. Note that the table `lr_table_in` is automatically propagated from Left to Right.

Figure 4: AG-driven disambiguation

In our Figure 3 example, each ambiguity branch of the ambiguity will be evaluated. The interpretation of `printf` is one such ambiguity according to the rules of Figure 4. During the traversal of the `stdio.h` part, `printf` will be recorded in `lr_table_in` as declaring a `Function`; during the traversal of the `main` part, the disambiguation rules of Figure 4 will therefore flag as valid the derivation `Identifier` → `PrimaryExpression`, and *invalid* the one with `Identifier` → `EnumerationConstant` because the lookup for `printf` in`lr_table_in` will not match the `Enum` kind.

A small auxiliary tool will prune all the invalid options afterward, yielding a unique parse tree. This parse tree (which includes all the details about the layout, comments, exact characters that were used etc.) is then simplified into a much more compact AST (freed from the layout, lexical details, etc.).

3.4 Transformations

The C-Transformers project, and its peer C++-Transformers, is somewhat ill-named, since it does not directly address transformation; rather it is a *workshop for implementing C program transformations*. Any transformation system is suitable, provided it supports our format for ASTs.

Amongst the possible engines to express transformations, we particularly like the Stratego programming language [1]. Because Section 4.3 demonstrates a transformation written in Stratego, it is worth being described here.

In Stratego every piece of data is a **term**, i.e., a (abstract syntax) tree. **Conditional rewriting rules** specify how a particular tree matching a specific shape should be transformed. A specific transformation, such as translating extended C to C, involves several rewriting rules to apply at different places of the tree, and in a specific order. **Rewriting strategies** provide an elegant means to control when and where to apply rewriting rules. In addition, a rich set of operators allows to build arbitrarily complex transformations (i.e., strategies) from simple atomic ones.

Many transformations are sensitive to the static scoping rules of the target language. For in-

stance α-conversion, the renaming of variables, must assign different names to the same identifier occurring in different scopes. **Dynamic (rewriting) rules** handle scopes gracefully: they can be created at any time but have their existence bound by scopes. To perform α-conversion, traverse from left to right, and for each variable declaration create a rewriting rule that maps the identifier to a fresh one. Entries and exits of scopes trigger the creation and destruction of the associated dynamic rules. This is much simpler than having to write generic (static) rules dealing with tables of symbols.

Finally, thanks to a tight integration with SDF, Stratego features **concrete syntax**: although rewriting rules do transform *abstract* syntax trees, rules can be written in the target's language *concrete* syntax. Examples of Stratego are given in Section 4.3.

In the Figure 3 example, a transformation could be performed on the AST, e.g., replacing the call to `printf` by a call to the faster `fputs` function.

3.5 Pretty-Printing

Pretty-printing, i.e., conversion from an abstract syntax tree to concrete syntax (text), is performed by Generic Pretty-Printer (GPP) [10], driven by language specific tables. These tables are generated from the SDF grammar, with embedded handcrafted directives to improve the result.

In the Figure 3 example, the final pretty-printed result of our chain would be *very* different from its input since instead of `#include <stdio.h>` one would have the whole content of the file. This issue is discussed in Section 6.

4 Case Study

C-Transformers provides a simple and powerful framework to transform C programs. To demonstrate its capabilities, we extend C into ContractC: C with "design by contract" support. The C-Transformers framework will be used to compile ContractC down to ISO C.

4.1 Design by Contract

Design by Contract is a software design and implementation methodology invented and promoted by Bertrand Meyer for his language, Eiffel [23]. The starting point is to consider that a function call involves two parties, the supplier and the client. The signature of the function is a (weak) form of a contract:

- the types of the incoming argument(s) are requirements put on the client (the caller) by the supplier (the callee);
- the type of the outgoing result(s) are guarantees given to the client by the supplier.

A successful function call requires that both parties respect their part of the contract. Statically typed programming languages enable statical checks, i.e., at compile time, while dynamically typed languages delay the verification until execution time. Note that in addition the signature of the function is a (weak) form of documentation. For instance, the signature of the square-root function, `double -> double`, specifies that it requires and returns a floating point number. Such information is always provided either in the documentation, or in comments, in dynamically typed languages.

Design by Contract extends signatures to include predicates between incoming and outgoing arguments. For instance the square-root function requires a non negative argument (a **precondition**) and ensures that the square of its result equals the incoming argument (a **postcondition**). Support for and use of pre-/postconditions dramatically improve the safety of programs, in particular when reusing components [12] — as an extreme example, Eiffel promoters claim that design by contract could have avoided the failure of the Ariane 501 launcher [14].

```
double sqrt (double r)
    precondition
    {
        r >= 0.;
    }
    postcondition (result)
    {
        result >= 0.;
        equal_within_precision (result * result, r);
    };
```

Figure 5: An function declaration example in ContractC

```
module PrePostConditions

imports Declarators

exports
  sorts
    DirectDeclarator ReturnValueDeclaration
    Assertion PostCondition PreCondition

  context-free syntax
    DirectDeclarator "(" ParameterTypeList ")"
      PreCondition? PostCondition?  → DirectDeclarator
```

This rule is based on [9, Section 2.3].

Figure 6: Extension of the C grammar to support pre- and postconditions

To demonstrate the use of C-Transformers, in the following we add support for pre- and postconditions to the ISO standard of C, based on the proposal of [9] for a C++ standard extension. The resulting language is here named ContractC.

4.2 Syntax

The adaption of the C++ extension proposal [9] to C results in adding support for pre- and postconditions to function *declarations*, not implementation. Indeed, since pre- and postconditions are pieces of formal documentation, they belong to the header file, which corresponds to the interface of a module in C parlance. Nevertheless, when compiled, the contract is to be integrated in the implementation of the function: it is the callee which will ensure that pre- and postconditions are properly met.

See Figure 5 for an example of ContractC: a set of pre- and postconditions put on the function `sqrt`.

To implement ContractC in Transformers, the first step is to extend its grammar with an additional rule for function declarations: Figure 6.

4.3 Compilation to C

The ContractC compiler (towards C) translates the contract into code run by the supplier — the function called. As an example, the ContractC declaration of Figure 5 transforms the regular C implementation of Figure 7 into that Figure 8. Writing such a transformation in Stratego is simple: see Figure 9, Figure 10, and Figure 11.

```
double sqrt (double r)
{
  return _libc_sqrt (r);
}
```

Figure 7: A C function implementation

```
double sqrt (double r)
{
  double result;
  {
    assert(r >= 0.);
  }
  {
    result = _libc_sqrt (r));
    goto end;
  }
end:
  {
    assert (result >= 0.);
    assert equal_within_precision (result * result, r);
  }
  return result;
}
```

This is the final result, when the contract specified in the function declaration (Figure 5) is inserted in the body of the function implementation (Figure 7).

Figure 8: A C function implementation with contracts installed

```
prepost = io-wrap(alltd(FunDecl <+ Contract))
```

In a single top-down traversal (`alltd`), for each function declaration with contract create a rule to install the contract, or for each function definition, install the contract.

Figure 9: The top-level of the transformation

```
FunDecl:
  Decl|[ ret fn (args) pre post ; ]| ->
  Decl|[ ret fn (args) ; ]|
    where
      rules (Contract :
        FunDef|[ ret' fn (args') cpstm ]| ->
        FunDef|[ ret' fn (args') cpstm' ]|
          where <transform(|ret', pre, post)> cpstm => cpstm')
```

Each function declaration with a contract must be rewritten without, and create an additional instance of `InstallContract` dedicated to the current function name `fn`.

Figure 10: Handling a ContractC function declaration

```
InstallContract(|as1, as2, type, res):
  CmpdStm|[ body ]| ->
  CmpdStm|[
     {
       type res;
       {
         as1
       }
       body'
     end:
       {
         as2
       }
       return res;
    }
  ]|
    where <alltd(ReturnToGoto(|res))> body => body'
```

Installing the contract takes more code to take several details into account: pass conditions to `assert`, gather the return type to declare the `result` variable, handle possible name clashes with its name, replace `return` with assignment-and-goto etc. Ultimately, once converted the pre-/postconditions assertions `as1`/`as2`, the `return` in the `body`, and the name of the result variable `res`, the function declaration is transformed.

Figure 11: Installing the contract in the function implementation

5 Discussion

To be effective, the C-Transformers tool chain must be easily extensible, i.e., every component must be easily configurable. In other words, featuring modularity is not merely satisfying with regards to current programming mottos, it is a must-have for every single tool involved in the chain.

In order to support **modularity**, the *parsing technique* virtually needs to support the full class of context-free grammars — for usual proper subclasses such as LL and LR are known not to be stable under union. The scanner-less generalized LR parser, SGLR supports the full context-free class of grammars — and actually some more. In addition SDF provides powerful and convenient means to compose modules. The built-in support for (possibly ambiguous) AST construction also enable to focus on actual issues, instead of having to code lengthy AST support classes or to fight parser conflicts.

The *disambiguation* also requires modularity in its strongest sense: a simple disjunctive "union" of disambiguation tools won't suffice. For instance, the sample ambiguities cited in the introduction, `(a) * (b)`, obviously affect the pre- and postconditions. Not only do we need to be able to add new disambiguation rules to those of the host language, but we also need to intermix them — just as freely as for SGLR modules. Attribute grammars handle this gracefully, being modular by nature, and better yet, sharing the exact same definition of modularity as the grammar itself.

The *transformation* needs to focus on specific spots; it is not concerned by most of the nodes. Stratego provides generic traversal operators that not only relieve the programmer from tedious work, but also guarantee the independence of the transformation from changes in the host grammar. In other words, Stratego also meets the modularity requirements.

Finally the *pretty printing engine* needs to support plug-ins to express the visual structure of the additional constructs. While GPP does support such add-ons, we find embedded pretty-printing rules more convenient. Being bound to the grammar, they share its modularity in the exact same sense, like attribute rules do.

If any of these components were to lack modularity support, or even slightly deviates from

Type	Length	Comment
Grammar	6 rules	Pre-/postconditions, function declaration
Disambiguation	6 lines	Same as for regular function declaration
Pretty Printing	4 rules	Pre-/postconditions, function declaration
Meta-variables	3 rules	Pre-/postconditions, assertions
Transformation	60 lines	20 of which obey C formatting rules

Figure 12: ContractC implementation effort

the standard set by SDF and SGLR, then the transformation might become undoable, or would require massive infrastructure. The result would also become extremely fragile: any change in the host grammar or in the extension can possibly require a full overhaul. C-Transformers features the full concept of modularity, allowing concise and robust implementations of transformations.

Although still young, C-Transformer is already very **usable**: Thanks to modularity the implementation of ContractC is quite straightforward and very compact. The volume of the full project is given in Figure 12.

The whole processing chain is composed of several steps, each one adding its own overhead to the actual transformations.

Preprocessing We run a home-grown preprocessor, equivalent to POSIX `cpp`, but producing slices of the input.

Parsing The execution of SGLR on each slice.

Concatenation All the different slices are pasted together.

Evaluation The attributes are computed on the parse tree.

Pruning Removal of the alternatives of ambiguities flagged as incorrect by the evaluation.

Checking Making sure no ambiguity remains.

Implosion Conversion to a suitable form for transformations (construction of an AST from the parse tree).

Transformation Compilation of ContractC to C.

Pretty-printing The AST is converted back to a concrete syntax program.

Three examples were chosen to measure the contributions of each step, see Figure 13.

The great variation of the figures is due to the fact that these samples are quite small, the last one being the only significant example. Then the whole processing is dominated by the conversion of the parse tree in an AST. We conclude that the cost of our technology AG-driven disambiguation is negligible, and there is not even reasons to try to optimize it further. Unfortunately we also conclude that, currently, parsing ambiguously grammars as ambiguous as those of C and C++ and then disambiguating is somewhat prohibitive.

We have been told that future versions of SGLR might perform the implosion during the parsing. Maybe some of the cost will be lowered, but unless SGLR is also able to run AG-driven disambiguation, it will still have to build massive (ambiguous) ASTs that will be pruned afterward.

	Queens		Hello, World		Lemon	
Lines of code	76		448		3550	
Duration	s	%	s	%	s	%
Preprocessing	0.11	15	0.13	1	0.18	0
Parsing	0.09	12	2.41	25	8.11	4
Concatenation	0.01	1	1.3	13	2.73	1
Evaluation	0.12	16	1.14	12	14.16	6
Pruning	0.08	11	0.8	8	8.15	4
Checking	0.01	1	0.2	2	1.53	1
Implosion	0.13	18	3.15	33	179.5	81
Transformation	0	0	0.02	0	0.35	0
Pretty-printing	0.18	25	0.53	5	6.95	3
Total	0.73	100	9.68	100	211.66	100

"Queens" is extremely short and includes no header. "Hello, World" is presented in Figure 3. "Lemon" is a parser generator that fits in a single C file.

Figure 13: Running time of the C-Transformers chain

6 Conclusion

We have presented modern program transformation techniques using C-Transformers as an example. To demonstrate the intrinsic power of these techniques, we implemented an extension to ISO C in an extremely reduced number of lines without sacrificing readability. We have emphasized how essential modular and language generic tools are.

In the future, several issues deserve more work.

The current AG system works perfectly well with very satisfying performance, but many improvements are expected. The most notable addition will be... additional syntactic sugar to cover the most common idioms we find during the implementation of semantics driven disambiguation filters: currently the attribute rules are lengthy and repetitive. Up to date AG engines, e.g. Utrecht University Attribute Grammar System (UU-AG) [2], abstract attributes from production rules, which allows shorter and more readable declarations. Because our AGs are written in SDF and transformed with Stratego/XT, the methodology described in this paper can be applied to them.

The most severe issue with C-Transformers is that it "disrupts the programming style" [33]: it processes and produces *preprocessed* C, i.e., with all the `#include` and other `#define` expanded. For instance, the simple "hello world" 4 liner program (see Figure 3) actually contains 450 lines of code coming from `#include <stdio.h>`. Not only does this clutter the result, it also makes it non portable. Indeed, much of the portability of C is handled by system headers that make wide use of operating system and/or architecture-specific code. This issue can be a show stopper to some possible applications of the C-Transformers project, for instance when users wish to ship the product but not the producer, or when the result is meant to be maintained by humans — as opposed to pre-processing phases. This limitation is a deliberate *temporary* choice: our limited resources were assigned to address the disambiguation first, and *then* programming style will be preservable. To cope with this issue we planned to develop a reversible preprocessor that embeds annotations in the AST to allow their reversal, very much in the spirit of Proteus project in fact [33].

Once C completely tamed, we will focus (again) on C++. C++ inherits ambiguities from C such as `(a) * (b)`, but it also adds ambiguities of its own, even when identifier kinds are known. 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)`. The `template` mechanism makes the process complex ad nauseam, requiring not only to carry symbols in tables, but also arbitrarily long ASTs!

Our C++ grammar is complete and the disambiguation filter is almost completed. In a foreseeable future, also making use of the reversible preprocessor, developing useful transformations with C++-Transformers should be possible.

7 Acknowledgments

The Transformers was started by Robert Anisko while an LRDE student; today it is developed by other LRDE students under the supervision of Akim Demaille. Valentin David was the architect of the current disambiguation chain (see Section 3.3), now maintained by Alexandre Borghi. Other LRDE students have contributed significant portions of the Transformers project: Clément Vasseur, Nicolas Pouillard, and Olivier Gournet. The authors thank particularly Olivier Gournet who, behind the scene, made ContractC work.

8 Biography

Alexandre Borghi is an LRDE student. He works on AG driven disambiguation as the current maintainer of the Transformers disambiguation chain.

Valentin David is a former LRDE student working specially on C++ parsing in the Transformers project. He is now a Ph.D. student at the University of Bergen, working on C++ transformation.

Akim Demaille is the director of the EPITA Research and Development Laboratory (LRDE). He teaches language theory, compiler construction, and formal logic at EPITA. His research topics include program transformation with the Transformers project [19], automata theory with the Vaucanson project [22], and compiler construction pedagogy with the Tiger project [21].

References

[1] www.stratego-language.org.

[2] Arthur Baars, Doaitse Swierstra, and Andres Löh. Utrecht University Attribute Grammar System, 1999. http://www.cs.uu.nl/wiki/Center/AttributeGrammarSystem.

[3] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65–74, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[4] Mark van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[5] Mark van den Brand, Steven Klusener, Leon Moonen, and Jurgen J. Vinju. Generalized parsing and term rewriting: Semantics driven disambiguation. volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

[6] Martin Bravenboer, René de Groot, and Eelco Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, Braga, Portugal, July 2005.

[7] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[8] Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss. A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL)*, Anaheim, CA, USA, October 2003.

[9] Lawrence Crowl and Thorsten Ottosen. Proposal to add contract programming to C++ (revision 3). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1866.html, August 2005.

[10] Merijn de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[11] Alexandre Duret-Lutz. Olena: a component-based platform for image processing, mixing generic, generative and OO, programming. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—-Young Researchers Workshop; published in "Net.ObjectDays2000"*, pages 653–659, Erfurt, Germany, October 2000.

[12] ISE Software. Building bug-free O-O software: An introduction to design by contract(TM). http://archive.eiffel.com/doc/manuals/technology/contract/page.html, 1993.

[13] ISO/IEC. ISO/IEC 9899:1999 (E). *Programming languages - C*, 1999.

[14] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997. http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html.

[15] Merijn de Jonge, Eelco Visser, and Joost Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001.

[16] Merijn de Jonge and Joost Visser. Grammars as contracts. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99, Erfurt, Germany, October 2001. Springer.

[17] Donald E. Knuth. Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127–145, 1968.

[18] Sylvain Lombardy, Raphaël Poss, Yann Régis-Gianas, and Jacques Sakarovitch. Introducing Vaucanson. In Springer-Verlag, editor, *Proceedings of Implementation and Application of Automata, 8th International Conference (CIAA)*, volume 2759 of *Lecture Notes in Computer Science Series*, pages 96–107, Santa Barbara, CA, USA, July 2003.

[19] LRDE — EPITA Research and Developpement Laboratory. Transformers home page, 2005. http://transformers.lrde.epita.fr.

[20] LRDE. Olena home page, 1999. http://olena.lrde.epita.fr/.

[21] LRDE. Tiger project home page, 2000. http://tiger.lrde.epita.fr/.

[22] LRDE. Vaucanson home page, 2001. http://vaucanson.lrde.epita.fr/.

[23] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, March 2000.

[24] George C. Necula. CIL home page, 2005. http://sourceforge.net/projects/cil.

[25] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[26] Karina Olmos and Eelco Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.

[27] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[28] Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, July 1997.

[29] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[30] Eelco Visser. A bootstrapped compiler for strategies (extended abstract). In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 73–83, Trento, Italy, July 5 1999.

[31] Eelco Visser. Tiger in Stratego, 2002. http://www.program-transformation.org/Tiger/WebHome.

[32] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[33] D. G. Waddington and B. Yao. High fidelity C++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, Electronic Notes in Theoretical Computer Science, Edinburgh University, UK, April 3 2005.

[34] Mikal Ziane. Towards tool support for design patterns using program transformations. In *Langages et Modèles à Objets (LMO)*, volume 7, pages 199–214, Le Croisic, January 2001. Hermès Science Publications.