A Source-To-Source Architecture for User-Defined Optimizations

Markus Schordan[1] and Dan Quinlan[1]

Lawrence Livermore National Laboratory, CA 94551, USA,
schordan1@llnl.gov, dquinlan@llnl.gov

Abstract. We present an architecture for the specification of source-to-source transformations. New source code can be specified as source-fragments. The translation of source-fragments to the intermediate representation is accomplished by invoking the frontend. For any inserted fragment we can guarantee that it is typed correctly. If no error is reported on inserted fragments, the whole program can always be compiled without errors. Based on a given abstract attribute grammar the user can specify transformations as semantic actions and can combine the computation of attributes with restructure operations on the intermediate representation.

1 Introduction

The development of special purpose (domain-specific) libraries to encapsulate the complexity of software is a significant step toward the simplification of software. But the abstractions presented by such libraries are user-defined and not optimized by the vendor's language compiler. The economics and maturation of new language and compiler designs make it particularly difficult for highly specialized languages to appear and be accepted by developers of large scale applications. Unfortunately, the generally poor level of optimization of user-defined abstractions within applications thus negates their effective widespread use in fields where high performance is a necessity.

Though significant aspects of our approach are language independent, our research work has targeted the optimization of C++ applications. The framework developed to support this research, ROSE, [1], allows us to express optimizations based on an abstract C++ grammar, eliminating the syntactical idiosynchrases of C++ in the specification of a transformation. Because we target library developers generally, our approach avoids the requirement that users learn a new special purpose language to express transformations. The semantic actions which specify a transformation are implemented in C++.

Within previous research we have demonstrated the use of ROSE [1, 2] and that the performance penalty of user-defined abstractions can be overcome by source-to-source transformations. We presented how a speedup of up to four can be achieved for user-defined abstractions as they are used in practice. The use of the semantics of the user-defined abstractions has been an essential part of this success. In this paper we demonstrate the use of the abstract grammar in combination with source strings and restructuring of the intermediate representation. As example we discuss the core of an OpenMP parallelization. The high-level semantics of the user-defined type utilized in the example is the thread-safety of its methods.

In section 2 we describe the architecture and how we can translate incomplete source-fragments to corresponding fragments of the intermediate representation. In section 3 we discuss how program transformations are specified by the use of the abstract grammar and source-strings. In the final sections we discuss related research and our conclusions.

2 Source-To-Source Architecture

In a usual source-to-source translation the frontend is invoked once. Transformations are either syntax directed or defined as explicit operations on an intermediate representation (IR). Eventually the backend is called to generate the final source program. In our architecture, see fig. 1, the frontend and backend are components that can be invoked at any point in an operation on the IR to obtain program fragments.

The capability of translating source-fragments to IR-fragments and back is essential to allow a compact specification of transformations as demonstrated in the example in section 3. This allows the definition of a transformation by combining sequential strings although our intermediate representation has a tree structure. Although strings are used, by invoking the frontend each fragment is type-checked before it is inserted in the IR. This ensures that in each step of a transformation, when a part of the intermediate representation is replaced by a new fragment, the program fragment is checked for syntactical and semantical correctness.

The combination of different source-fragments is specified in semantic actions associated with rules of an abstract grammar. The computed attribute values can be of arbitrary type, including source-fragments. Because the computed attributes can also be source-fragments it is necessary to translate them to IR-fragments to insert them into the IR. Note, we do not re-parse source-strings, a source-string is only parsed once by the frontend. But the frontend can be invoked to translate computed source-strings, ensuring that all semantic checks are performed on the inserted IR-fragments as well. Note that our approach does not require any modifications to an existing frontend.

We use the EDG-frontend [3] for parsing C++ programs. This frontend performs template instantiation and a full type evaluation. In our abstract grammar all type information is made available to the user as annotations of nodes in the abstract syntax tree (AST) which can be accessed in semantic actions of the abstract attribute grammar. The availability of semantic compile-time information is an essential aspect of our architecture. In the following sections we describe in detail how source-fragments can be translated to IR-fragments by utilizing an existing frontend and how all semantic information can be updated in the IR.

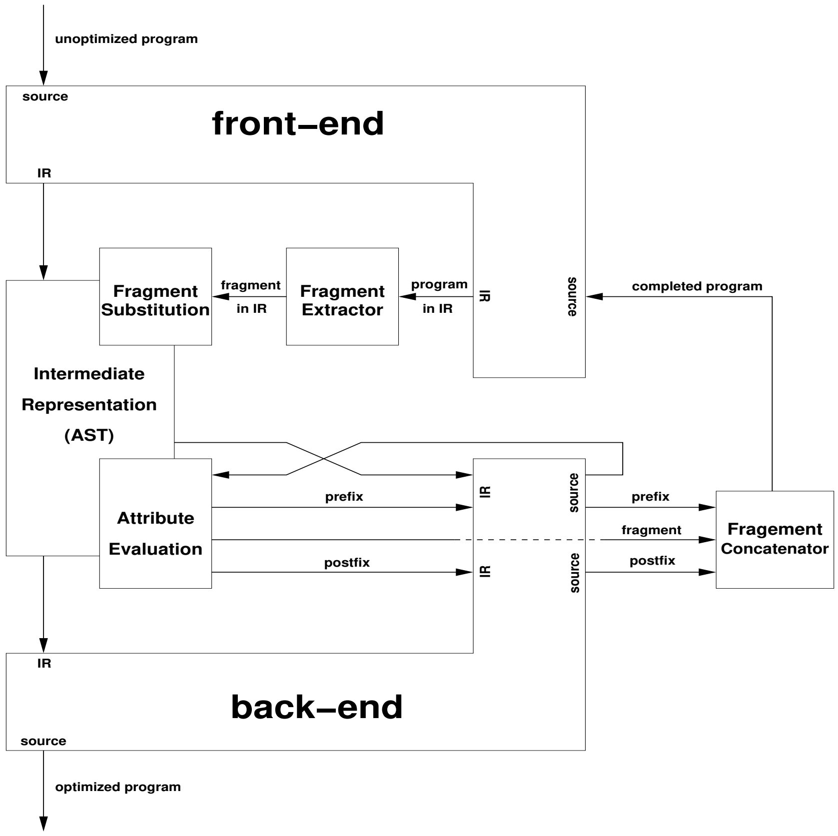

Fig. 1. Source-To-Source architecture with frontend/backend invocation

2.1 Fragment Concatenator and Extractor

In general, a source-fragment cannot be parsed by the frontend because it is an incomplete program. Therefore it needs to be extended by a source-prefix and a source-postfix to a complete program such that it can be parsed by the frontend. This computation of the prefix and postfix is automated. The user only specifies the fragment and the target location of the corresponding IR-fragment. In our IR, the target location, L_{abs}, is a node in the AST. The prefix and postfix are automatically generated. The source-prefix consists of all declarations and opening braces of scopes before the target location, the source-postfix consists of all closing braces of scopes after the target position.

The frontend returns a program in IR. From this the corresponding IR-fragment needs to be extracted. A source string shall be denoted as S and an intermediate representation as I. We shall denote any prefix by $\triangleleft$, any fragment by $\Box$, and any postfix by $\triangleright$.

A given source-fragment, $S_\Box$, is translated to an IR-fragment, $I_\Box$, by invoking the frontend.

The fragment concatenator concatenates the source-prefix $S_\triangleleft$, the source-fragment $S_\Box$, and the source-postfix $S_\triangleright$. Information necessary to extract the IR-fragment, $I_\Box$, corresponding to the source-fragment, $S_\Box$, from the IR of the completed program, shall be denoted L_{sep}. It represents separators that are inserted by the concatenator before invoking the frontend, and used by the extractor to separate the fragment from the prefix and postfix.

$$(S, L_{sep}) = \text{concatenator}(S_\triangleleft, S_\Box, S_\triangleright)$$

The completed program S can be parsed by the frontend

$$I = \text{frontend}(S)$$

to obtain the program in intermediate representation I. From this program I, the IR-fragment, $I_\Box$, is extracted by the fragment extractor.

$$I_\Box = \text{extractor}(I, L_{sep})$$

The fragment extractor strips off the IR-prefix, $I_\triangleleft$, corresponding to $S_\triangleleft$ and $I_\triangleright$ corresponding to $S_\triangleright$. Information on where these parts are separated, L_{sep}, which is returned by the fragment concatenator, is used to find start and end points of $I_\triangleleft$ and $I_\triangleright$.

We have shown how we can obtain the corresponding IR-fragment $I_\Box$ for a given source-fragment $S_\Box$ by invoking the frontend. The inverse operation, by invoking the backend, is

$$S_\Box = \text{backend}(I_\Box).$$

Since both representations, $I_\Box$ and $S_\Box$, can always be translated one to the other, both can be used interchangeably in the definition of a transformation.

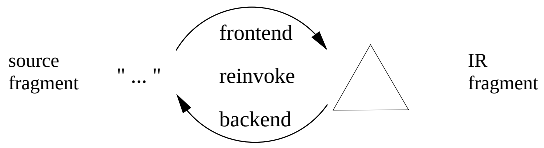

Fig. 2. A source-fragment can always be translated into an IR-fragment by invoking the frontend and an IR-fragment can always be translated into a source-fragment by invoking the backend.

In figure 2 this correspondence is shown as a diagram. The definition of a transformation is simplified because source-fragments, $S_\Box$, can be used to define source code patterns as strings. On the other hand, source-fragments corresponding to subtrees of the IR can always be used as values in an attribute evaluation because we can always obtain the corresponding source-fragment for an IR-fragment.

This allows the definition of a transformation by combining sequential strings although the intermediate representation has a tree structure. All semantic information, such as type information for each expression, symbol tables, etc., is updated by the underlying system. Note that the order in which the IR is processed is (mostly) source sequence.

2.2 Fragment Substitution

An IR-fragment, $I_\Box$, which is obtained from the fragment extractor, substitutes an IR-fragment in the IR as specified by L_{abs}.

$$I^{i+1} = \theta(\langle I^i_\Box, I_\Box, L_{abs}\rangle, I^i)$$

The substitution θ replaces the IR-fragment $I^i_\Box$ by the new IR-fragment $I_\Box$ (which corresponds to a $S_\Box$) at the specified location L_{abs}. $I^i_\Box$ is an IR-fragment in I^i. After a substitution has been applied the restructured IR, I^{i+1}, becomes accessible for the next transformation. This ensures that a substitution operates as a side-effect free function, with respect to the IR structure, in a transformation.

Once an attribute evaluation has been performed and a transformation is finished, I^{i+1} becomes accessible and I^i is no longer accessible. Note that fragments $I^i_\Box$ and $I_\Box$ can be empty, corresponding to empty strings ϵ (source-fragments), which allows to define insertions and deletions.

3 Program Transformations

Program transformations are specified as semantic actions of the abstract C++ grammar. The abstract grammar covers full C++. We use a successor of Coco/R [4], the C/C++ version ported by Frankie Arzu. Coco/R is a compiler generator that allows to specify a scanner and a parser in EBNF for context free languages. The grammar has to be LL(1). We use this tool to operate on the token stream of AST nodes. Therefore we do not use the scanner generator capabilities of Coco/R and implemented a scanner to operate on a token stream of AST nodes.

A terminal in our default abstract grammar always directly corresponds to AST nodes of one type. The name of this type is the name of the terminal in the grammar. The grammar can be modified but the user has to ensure that it still accepts all programs that are to be transformed. Our present version of the default abstract grammar for full C++ has 165 rules. Non-terminals either directly match names of base types in the AST's object-oriented class hierarchy, or the non-terminals were introduced (with the postfix NT in our default grammar) for better readability. The user can also access all annotated AST information gathered by the frontend at each AST node through a variable `astNode`. The variable always holds the pointer to the corresponding AST node of a parsed terminal.

Before transformation

```
for(ValContainer::iterator i=l.begin(); i!=l.end(); i++) {
   a.update(*i);
}
```

After transformation

```
#pragma omp parallel for
for(int i = 0; i < l.size(); i++) {
  a.update(l[i]);
}
```

Fig. 3. An iteration on a user-defined container `l` that provides an iterator interface. The object `a` is an instance of the user-defined class `Range`. Object `l` is of type `ValContainer`. In the optimization the iterator is replaced by code conforming to the required canonical form of an OpenMP parallel for. The user-defined method update is thread-safe. This semantic information is used in the transformation.

In the example source in fig. 3 we show an iteration on a user-defined container with an iterator. This pattern is frequently used in applications using C++98 standard container classes. The object `a` is an instance of the user-defined class `Range`. The transformation we present takes into account the semantics of the type `ValContainer` and the semantics of class `Range`. The transformation is therefore specific to these classes and its semantics.

For the type `ValContainer` we know that the type `iterator` defined in the class follows the iterator pattern as used in the C++98 standard library. For the type `Range` we know that the method `update` is thread safe. We show the core of a transformation to transform the code into the canonical form of a for-loop as required by the OpenMP standard. We also introduce the OpenMP pragma directive. Note that the variable `i` in the transformed code is implicitly private according to the OpenMP standard 2.0 . If the generated code is compiled with an OpenMP compiler, different threads are used for executing the body of the for-loop. The test, `isUserDefIteratorForStatement`, to determine whether the transformation can be applied, is conservative. It might not always allow to perform the optimization although it would be correct but it is never applied when we cannot ensure that the transformed code would be correct.

In the example in fig. 4 the rule of `SgScopeStatement` is shown. The terminal `SgForStatement` corresponds to an AST node of type `SgForStatement`. The variable `astNode` is a pointer to the respective AST node of the terminal and assigned by our supporting system when the scanner accesses the token stream. Note that every terminal in the grammar corresponds to a node in the AST, except the parentheses.

Methods of the object `subst` allow to insert new source code and delete subtrees in the AST. The substitution object `subst` buffers pairs of target location and string. The substitution is not performed before the semantic actions of all subtrees of the target location node have been performed. This mechanism allows to check whether substitutions would operate on overlapping subtrees of the AST (in the same attribute evaluation). In case of overlapping subtrees an error is reported.

The object `query` is of type `AstQuery` and provides frequently used methods for obtaining information stored in annotations of the AST. These methods are also implemented as attribute evaluations.

```
SgScopeStatement<bool isOmpFor>
  = SgForStatement
      (.
        isOmpFor
        = ompTransUtil.isUserDefIteratorForStatement(astNode,isOmpFor);
      .)
    "(" SgForInitStatementNT<isOmpFor> SgExpressionRootNT
        SgExpressionRootNT SgBasicBlockNT<isOmpFor>
    ")"
      (.
        if(isOmpFor) {
          string ivarName = query.iteratorVariableName(astNode);
          string icontName = query.iteratorContainerName(astNode);
          string modifiedBodyString
                 = ompTransUtil.derefToIndexBody(ivarName,icontName);
          string beforeForStmt
                 = "#pragma omp parallel for\n";
          string newForStmt = "for( int "+ivarName+"=0;"
                              + ivarName+"<"+icontName+".size();"
                              + ivarName+"++ ) "+modifiedBodyString;
          subst.replace(astNode,beforeForStmt + newForStmt);
        }
      .)
  | ...
```

Fig. 4. A part of the SgScopeStatement rule of the abstract C++ grammar with the semantic action specifying the transformation of a SgForStatement.

The inherited attribute `isOmpFor` is used to handle the nesting of for-loops. It depends on how an OpenMP compiler supports nested parallelism whether we want to parallelize inner for statements or only the outer for statement. In future this decision will be made more specific to OpenMP compilers on different platforms and the boolean attribute will be replaced by an object to provide more information about the context of OpenMP for-loops.

The object `query` of type `AstQuery` offers methods to provide information on subtrees that have been proven to be useful in different transformations. In the example we use it to obtain the name of the iterator variable, and to obtain the node of the declaration of the iterator variable. Note that these functions must return valid values because it has been tested that the for-loop qualifies for transformation before.

The example shows how we can decompose different aspects of a transformation into separate attribute evaluations. The methods of the query object are implemented by using the attribute evaluation. For this reason we allow to call any method of the recursive descent parser generated by COCO to parse a sublanguage, and start an evaluation at a certain node in the AST. Multiple grammar files can also be used for such cases and each file contains a version of the abstract C++ grammar. In the example, `isUserDefIteratorForStatement` is a wrapper function of another attribute evaluation generated by COCO that starts at a SgForStatement node.

In fig. 3 the generated code is shown. The access uses the notation for random access iterators. Even if the access is not of complexity O(1) the parallelization can still provide speedup. The user who implements the transformation has to take such tradeoffs into account in a test function to decide whether a transformation should be applied or not. Note that the generated source code can have a slightly different formatting because the format of the source code is a beautified version of the source corresponding to the transformed AST.

4 Related Work

We use Sage III as intermediate representation, which we have developed as a revision of the Sage II [5] AST restructuring tool. Its predecessor, Sage++, included a Fortran frontend, while Sage II included the EDG C++ frontend [3] and represented a more robust handling of C++ as a direct result. Our work has substantially modified Sage II (e.g., adding template support and changes of the structure and interfaces of Sage II of about 25% of the node classes). Sage II required modifying the AST by explicitly rearranging pointers between AST nodes and creating new node objects if new code needed to be added. In our framework this can be done by using source strings and an abstract grammar.

Related work on the optimization of libraries on telescoping languages [**?**] shares many of the same goals as our research work and we expect to work more closely with these researchers in the near future. Our approach so far is less ambitious than the telescoping languages research, but is in some aspects further along, though currently specific to abstractions represented in C++.

Further approaches are based on the definition of library-specific annotation languages to guide optimizing source code transformations [6] and on the specification of both high-level languages and corresponding sets of axioms defining code optimizations, see [7] for example. We address the need of annotations for guiding optimizations either by pragmas, comments, or make optimizations specific to user-defined types as discussed in the example transformation.

Kimwitu [8] allows to associate semantic actions with rules of a tree grammar. Conceptually Kimwitu could be used instead of COCO as well. But the substitution mechanism is more difficult to integrate into our system when using the C++ version of Kimwitu from our experience because it uses its own memory handling and puts restrictions on some code fragments used in semantic actions. COCO was easier to integrate in our system because it only copes with issues of parsing and not transformation, and does not put any restrictions on the code used in semantic actions. However, our grammar conforms to the essential properties of a tree grammar as required by Kimwitu. The other mode of Kimwitu, to express term rewriting explicitly by using subterms describing subtrees on both sides of a rule, is an advantage of Kimwitu in particular for the compact specification of algebraic optimizations.

The Microsoft .NET CodeDom Compiler Framework is used by various tools, including ASP.NET and Visual Studio.NET. It offers an interface for restructuring source-code, designed to handle different languages. Nigel Perry has defined an abstract grammar for the CodeDom Language [9]. A program in the language is represented by a tree of CodeDom objects, which corresponds to a parse tree in a compiler for a conventional language. In our framework the abstract grammar can actually be used to specify transformations. In the abstract CodeDom EBNF grammar, type information is made explicit as extension in the grammar. In our grammar type information is available as accessible annotation of the AST nodes. Also we do not use tree extensions to identify grammar symbols that correspond to AST nodes. A terminal always directly corresponds to an AST node. We only added parentheses to the token stream. However, most of all information required for our approach is available for the CodeDom framework, which makes it an interesting target in our future work.

5 Conclusions and Future Work

The use of an abstract grammar greatly simplifies the specification of a source-to-source transformation. Many aspects of parsing source code and type evaluation are not helpful for expressing code transformations. The specification of a source-to-source transformation should not interfere with specific parsing issues of the concrete syntax of the language. On the other hand, the concrete syntax is what developers, who want to optimize their application codes, are most familiar with. From this we conclude that offering the use of source-strings for specifying new code and using an abstract grammar to allow to specify transformations is a practical solution to this problem. The availability of full type information is necessary for the optimization of user-defined abstractions.

Instead of requiring the user to learn a new language to express transformations, all transformations are themselves defined in C++, the same language in which the application code is written and which the user seeks to optimize. The grammar forces the user to structure the transformation according to the structure of the language, the decomposition in different transformation objects, as shown in the example, gives the necessary freedom in designing complex transformations.

Future work is targeted at demonstrating the development of a wide range of optimizing source-to-source translators for specific scientific libraries and applications. Additional work is the analysis of complex data structures to automate the generation of application specific tools (connection to visualization libraries, dump/restart functions, etc.).

By permitting developers to add highly tailored companion optimizations to their user-defined types and applications, we define a hierarchical (telescoping) approach to language design which builds incrementally upon existing general purpose languages. We hope that a similar approach could in the future form a significant mechanism within a general purpose language compiler to allow users to extend the range of optimizations.

References

1. Daniel Quinlan, Brian Miller, Bobby Philip, and Markus Schordan. Treating a user-defined parallel library as a domain-specific language. In *16th International Parallel and Distributed Processing Symposium (IPDPS, IPPS, SPDP)*, pages 105–114. IEEE, April 2002.
2. Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 2003, to appear.
3. Edison Design Group. http://www.edg.com.
4. Hanspeter Moessenboeck. Coco/R - A generator for production quality compilers. In *LNCS477, Springer*, 1991.
5. Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.
6. Samuel Z. Guyer and Calvin Liri. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 39–52, Berkeley, CA, October 3–5 1999. USENIX Association.
7. Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *Conference Proceedings of the 1999 International Conference on Supercomputing*, pages 434–443, Rhodes, Greece, June 20–25, 1999. ACM SIGARCH.
8. P. van Eijk, A. Belinfante, H. Eertink, and H. Albas. The term processor Kimwitu. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 96–111, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.
9. Nigel Perry. A definition of the codedom abstract language, http://www.mondrian-script.org/codedom/codedom grammar.html, 2002.