Automatic Generation of Code Analysis Tools: The CastQL Approach

Christakis Lezos, Grigoris Dimitroulakos, Ioannis Latifis, Konstantinos Masselos
University of Peloponnese, Department of Informatics and Telecommunications
Terma Karaiskaki, 22100 Tripoli, Greece
{lezos,dhmhgre,latifis,kmas}@uop.gr

ABSTRACT

Source code analysis and manipulation tools have become an essential part of software development processes. Automating the development of such tools can heavily reduce development time, effort and cost. This paper proposes a framework for the efficient development of code analysis software. A tool for automatically generating the front end of analysis tools for a given language grammar is proposed. The proposed approach can be applied to any language that can be described using the BNF notation. The proposed framework also provides a domain specific language to concisely express queries on the internal representation generated by the front end. This language tackles the problem of writing complex code in a general purpose programming language in order to retrieve information from the internal representation. The approach has been evaluated through two different realistic usage scenarios applied to a number of different benchmark applications. The front end generator has also been tested for twenty input grammars. In all cases the software generated by the proposed framework functions according to the input grammar while the development time has been reduced on average down to 12% compared to equivalent handwritten implementations. The experimental results give evidence that the use of the proposed framework can heavily reduce the relevant design effort and cost.

CCS Concepts

•Software and its engineering → Domain specific languages; Source code generation; Translator writing systems and compiler generators;

Keywords

Domain specific languages; code query languages; source code analysis; compiler generators

RWDSL '16, March 12 2016, Barcelona, Spain

ISBN 978-1-4503-4051-9/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2889420.2889422

1. INTRODUCTION

A plethora of source code analysis and manipulation tools, both academic and commercial, have been proposed, especially the last two decades. Such tools are part of developers' workflow and all major IDEs incorporate them as extensions for numerous tasks. They can be used for program comprehension, software reengineering, refactoring, program metrics extraction and debugging and for many other purposes.

A code analysis and manipulation tool typically includes a front end to parse the input language(s) and generate an *internal representation (IR)*, usually in the form of an *abstract syntax tree (AST)*. Many compiler frameworks provide front ends for one or multiple languages as well as a mechanism to access the IR. Using an existing framework would simplify things as the burden of developing the front end and designing the IR would not lie on the developer. Roslyn [31] and Clang [27] are good examples of interfaces employing this approach. However, the disadvantage of this approach is the limitation of working solely on the compiler's input languages.

Such tools also include a module for collecting program information from the IR. Existing compiler frameworks in many cases do not employ a querying system for this purpose and custom passes over the IR have to be applied instead. Complex code in a general purpose programming language would have to be written in order to develop these passes without the assistance of a querying mechanism. This is a considerable inconvenience since it is a time consuming task and a great overhead for the developer. Moreover, querying an IR can be clearly considered a domain specific problem. Modeling it with a *domain specific language (DSL)* would introduce better expressiveness and thus reduce the development time. DSLs are widely used to ease the modeling of problems on different scientific areas [18].

The work discussed in this paper targets the realization of software for the automated development of code analysis and manipulation tools in order to reduce the relevant design time, effort and costs. The proposed framework includes a number of cooperating components including:

- The *CastQL* DSL. A query language accompanied by the necessary steps for developing language dependent queries starting from the problem formulation and reaching the actual implementation. CastQL works on top of an AST representation called *contextual abstact syntax tree (cAST)*. It is an embedded (internal) DSL [18] that uses C++ as the host language. The design principles and notation patterns described in [18] and [20] are followed.

- An automatic front end generator (*FEgen*) producing a parser for any given grammar using the BNF notation. All the code required for the front end and cAST specification is automatically produced and no C++ coding is required from the developer. FEgen practically enables the use of CastQL on any input language.

Similar technologies are usually geared towards either analysis (examination oriented) [21, 2, 13, 12, 40, 6, 11, 3, 29, 9] or manipulation (transformation oriented) [39, 23, 7, 22, 14, 10]. The proposed framework targets the rapid development of tools for both areas. The following are the advantages of the proposed work:

- There is no limitation to a predefined set of input languages. The input specification is customizable using the BNF notation.
- CastQL's expressiveness simplifies code querying. Even in the occasion of a query that cannot be expressed in CastQL, an alternative exists: custom traversals can be developed directly in C++.
- The tools generated by the proposed framework can be distributed as standalone C++ applications. This leverages the need for a specialized execution environment that dominates most of the relevant works.

The remainder of this paper is organized as follows: An overview of the proposed framework is presented in section 2 while the CastQL language is described in section 3. Section 4 discusses the integration of the proposed work into the MEMSCOPT compiler as a demonstration example while in section 5 experimental results from different code analysis and manipulation tasks utilizing the proposed approach are presented. Section 6 presents a review of existing work in the field and section 7 discusses conclusions and directions for future work.

2. FRAMEWORK OVERVIEW

The proposed framework includes a number of different modules. The major ones are CastQL and FEgen. Figure 1 depicts the structure of the framework. Two main stages are included: (1) the front end generation phase, when the prototype of the tool is created by the FEgen tool, and (2) the development environment used afterwards to extend this prototype. These two phases are illustrated in Figure 1 as gray boxes entitled *FEgen* and *Tool Prototype* respectively. Arrows with solid lines are used to indicate the flow amongst the FEgen operations, arrows with dotted lines indicate transactions between the components of the generated tool, and the ones with dashed lines present the sequence of the various phases incorporated in the generated tool.

FEgen is utilized to parse a BNF grammar and derive the code that generates the parse tree. To achieve this, it produces an annotated version of the grammar in the form of standard Flex/Bison lexer and parser generators' .y and .l files. In the next step, two options exist for the specification of the AST representation:

- FEgen can automatically infer the AST representation from the parse tree. A pass is applied that refines the parse tree in the following ways: (1) recursive BNF constructs are transformed to a list of objects, and (2) intermediate grammar symbols that are part of chain productions are eliminated.

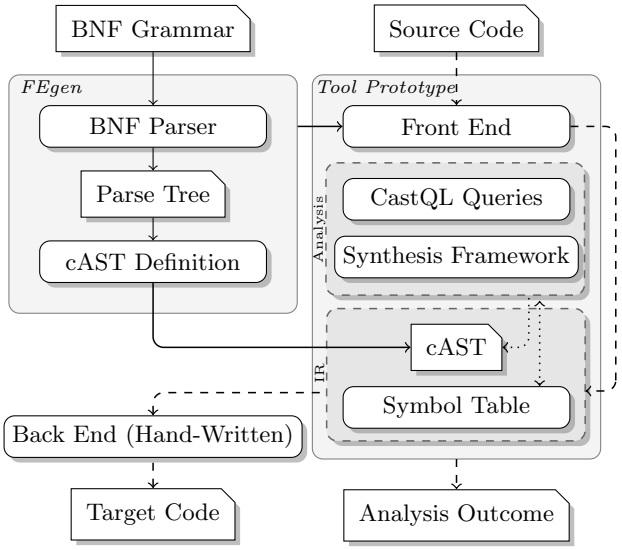


Figure 1: Overview of the proposed framework.

- A graphical interface is provided to the user for the custom definition of the AST constructs through annotations to the parse tree.

The source code generated by FEgen includes the front end of the tool and the internal representation specification. The IR includes the AST and a symbol table. FEgen cannot infer a symbol table for every input language. A basic template is provided which can be extended by the developer according to his needs.

On the second development phase, the developer uses the CastQL DSL and a library called *synthesis framework* to apply querying and manipulation operations on the AST. A modular way of query design is incorporated. Section 3 provides a description of the basic constructs and syntax of CastQL. The synthesis framework is a C++ library that assists the manipulation of the AST. It builds AST sub-trees utilized for code generation and refactoring operations.

An AST that simplifies code navigation and searching is used. *Contextual abstract syntax tree (cAST)* is a heterogeneous hierarchical tree whose nodes represent elements of the program and edges connecting them according to the input language's grammar rules. To encode the AST semantics and facilitate navigation and searching, each cAST element has zero or multiple contexts (according to its type) by which it connects to its descendants. The terms AST and cAST are used interchangeably for the rest of the paper.

3. THE CASTQL DESIGN

Hierarchical decomposition using a multilevel strategy is essential for tackling the problem of querying the code. Two kinds of queries depending on the level of abstraction are identified. *The high level queries (HLQs)* associate to the program level; they depend on the input language and retrieve information directly exploitable by following phases of the tool. To reduce complexity and maximize code reuse the HLQs are synthesized by a combination of *low level queries (LLQs)* that traverse the AST and collect nodes meeting specific criteria. The consecutive application of a specific combination of LLQs produces a set of AST nodes (*nodeset*)

that are the result of an HLQ. The information subsequently becomes available to the designer through the interface of the HLQ.

This method leaves the partitioning of information to HLQs to the designer. However, in the practical implementation example presented in section 4, relevant guidelines are given for the design of a query engine for the MEMSCOPT compiler. Figure 2 presents the proposed realization flow.

The HLQs are typical C++ classes implementing a specific interface while LLQs and nodesets are the basic constructs of the CastQL DSL. An LLQ is applied to a nodeset and always returns another nodeset as a result. There are several different types of LLQs available, as well as a number of search filters that can be applied on them. A short description of these filters and the LLQ types follows. The basic LLQ types include:

(1) node - collects the objects of a specific AST node type. Multiple node types can also be specified. **(2) context** - collects the objects residing in a specific AST node context. Multiple context types can also be specified. **(3) mixed** - nodes of a specific type and the ones residing in a specific context are both matched. **(4) name** - collects the object having a specific name. The symbol table prototype generated by FEgen must be utilized in order to take advantage of this LLQ type. **(5) complex** - a complex LLQ is created from multiple others and applies each of them sequentially to the input nodeset. **(6) similarity** - this LLQ collects the AST nodes that are the roots of sub-trees having the same structure as a given prototype sub-tree. **(7) setop** - returns the result of set operations applied on the input nodesets. **(8) clever** - this LLQ has the same behavior with a mixed LLQ but can also end the traversal upon reaching specific node or context types.

The search filters are essentially a parameter of the LLQs. They define which of the discovered nodes will be matched and when the query will terminate its operation. The basic search filters that can be applied on an LLQ include:

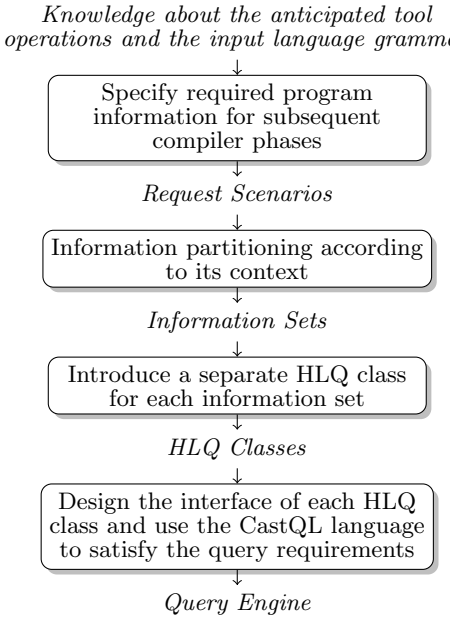


Figure 2: Proposed design implementation flow.

(1) shallow - discards the nodes discovered in AST locations other than the immediate descendants of the query starting points. **(2) deep first found** - search may reach at any depth but returns the first node matching the input. **(3) deep random access** - search may reach at any depth but returns the n-th AST node in the discovered sequence. **(4) exhaustive** - search may reach at any tree depth and returns the full set of nodes matching the input. **(5) depth specific first found** - search returns the first node matching the input located at most to a given depth. **(6) depth specific random access** - search may reach a given depth but returns the n-th node in the discovered sequence. **(7) depth specific exhaustive** - search returns the whole set of AST nodes matching the input located at most to a specific depth.

Another parameter of the LLQs concerns the order and direction of the AST traversal. DFS traversal is the default while BFS and upwards traversals are the alternative options. In upwards traversal the direction changes and the parent of each examined node is visited instead of its descendants.

4. USE CASES

The framework described in this paper has been utilized in a number of compiler construction and code analysis works [15, 25, 26, 37]. In this paper the use of the proposed framework for the development of MEMSCOPT [15] source code optimizer is discussed. In [25] a dynamic analysis tool for C applications that exposes the critical application's loops for memory hierarchy optimization is discussed. In [26] the tool presented in [25] is enhanced with features for data reuse distance analysis and source code transformation recommendations for temporal locality optimization. The proposed framework has also been applied for the development of a Scilab-to-C compiler [37].

4.1 MEMSCOPT Overview

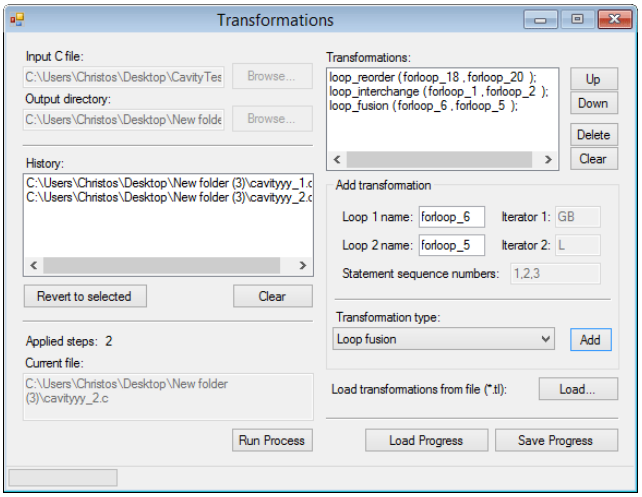
MEMSCOPT is an interactive source-to-source code optimization tool developed in the context of the FP7 ENOSYS project[1]. It acts on C programs compliant with the C89 standard. The tool has two operational modes, one for each of its two major facilities: 1) Analysis mode, and 2) transformations mode. In the analysis mode, the tool embeds the analysis results directly into the application code using a *special annotation language (SAL)*. SAL is used for both analysis assistance and documentation purposes. It is expressed inside C comments (Figure 3a) and the user can employ it to interact with the tool by providing input regarding analysis. Currently, SAL includes directives for loop naming, loop iterator identification, loop count and loop weight.

Optimization of the code is an iterative process where each step applies one or multiple refactorings. A large number of transformations can be applied by MEMSCOPT including *loop shift*, *loop extend*, *loop reversal*, *loop fusion*, *loop interchange*, *loop fission*, *loop normalization*, *loop reorder*, *loop switching*, *loopscopemoveforward*, and *loopscopemovebackward*. Figure 3b presents the interface for the transformation(s) mode. It includes fields for: 1) setting the input file and the output transformations' directory, 2) selecting a transformation from a transformation palette and configuring it, 3) executing, saving and restoring the transformation

[1]http://www.enosys-project.eu/

```
1 ...
2 /* @forloop:forloop_0 */
3 /* @iterator: k */
4 /* @#iterations: 5 */
5 /* @#loopweight: 1 */
6 for ( k=-GB; k<=GB; ++k ){
7   tot+=Gauss[abs(k)];
8 }
9 ...
```

(a)



(b)

Figure 3: a) Special annotation language. b) Transformations interface.

steps made so far or backtracking to specific steps, and 4) manipulating the transformations of each step. The whole number of transformation steps applied result in an XML transformation script. This script records the initial input file and the full details of each transformation step.

4.2 MEMSCOPT Implementation

In this section the query development process in MEMSCOPT is discussed. In the first design step, the information required by MEMSCOPT's analysis and optimization engines has been determined. The actual work in both cases concerns the C application's *for* loops which have to be identified and the determination of the information that must be retrieved. Table 1 lists the information required, the HLQ classes associated with this information, and a description of the corresponding query routines realized in each HLQ. The HLQ classes *HLQTranslationUnit*, *HLQFunctionDef*, and *HLQForLoop* have been introduced. The realization of these HLQs in CastQL is illustrated in Figures 4 and 5.

HLQTranslationUnit identifies the set of function definitions inside the input program. This functionality is realized in the CastQL code fragment of Figure 4a. It first creates an LLQ (lines 1-4) identifying function definition objects inside the AST. The exhaustive filter is applied which directs it to explore the whole set of functions in the program. Then, the query is applied to the root of the AST (lines 5-8). HLQTranslationUnit creates an HLQFunctionDef object for each function definition discovered in the program.

The objective of HLQFunctionDef is to identify the loops

Table 1: Information Request Scenarios.

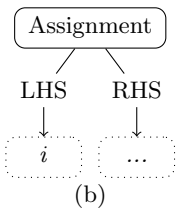
Request	Target HLQ
Get function definitions	HLQTranslationUnit
Returns the AST objects representing function definition	
Acquire function for-loops	HLQFunctionDefinition
Returns the AST objects representing the for-loops in the current function body	
Acquire nested for-loops	HLQForLoop
Returns the for-loops that nest in the body of the current loop	
Get loop body iterator instances	HLQForLoop
Returns the AST objects corresponding to the loop iterator in the context of the loop body	
Get initializer expression	HLQForLoop
Returns the right hand side of the assignment used to initialize the loop	
Get initialization expression	HLQForLoop
Returns the expression used to initialize the loop	
Get adjustment expression	HLQForLoop
Returns the expression used to update the iterator in each loop iteration	
Get finalizer expression type	HLQForLoop
Returns the operator type used to update the loop iterator	
Get finalization expression type	HLQForLoop
Returns the expression used to finalize the loop	
Get loop body statements	HLQForLoop
Returns the loop body statements	
Get primary iterator	HLQForLoop
Returns the primary induction variable of the loop	

lying in the scope of a specific function. This is realized in the CastQL code fragment of Figure 5a. Two distinct LLQs (lines 1-4 and 5-9) are combined in a single complex LLQ (lines 10-13). The first LLQ routes the search to the function body, while the second identifies the underlying loops. The latter applies the depth specific exhaustive filter to exclude the nested loops lying in depth greater than 1. The identified loops are deposited in the *seto1* nodeset for subsequent exploitation. A new HLQForLoop is instantiated for each discovered loop. These HLQForLoop objects conduct new searches to identify nested loops.

An HLQForLoop first identifies the loop's induction variables. A variable must appear in all four contexts of a loop (initialization, finalization, adjustment, and body) to classify as induction variable. Figure 5b presents the CastQL code that implements this. The *inter3* nodeset contains the variables appearing in all four regions. The expressions defining the loop's iteration space are also determined by HLQForLoop. It first looks for any assignment expressions containing the induction variable in the loop initialization region. Then, it identifies the expression that initializes the

```
1 fundefsPat = LLQ::DEFINE(NODE)
2   .nodetype(FunctionDef)
3   .filter(EXHAUSTIVE)
4 .END();
5 fundefsSet = LLQ::APPLY()
6   .query(fundefsPat)
7   .on(mASTRoot)
8 .END();
```

(a)



(b)

Figure 4: a) CastQL code fragment to identify function definition AST objects. b) Prototype AST subtree for matching induction variable initialization.

```
pat1 = LLQ::DEFINE(CONTEXT)
  .contexttype(FunctionDef_Body)
  .filter(DEEPFIRSTFOUND)
.END();
pat2 = LLQ::DEFINE(NODE)
  .nodetype(ForLoop)
  .filter(DEPTHSPECIFIC_EXHAUSTIVE)
    .depth(1)
.END();
```

(a)

```
patIdent = LLQ::DEFINE(NODE)
  .nodetype(Identifier)
.END();
patInitId = LLQ::DEFINE(CONTEXT)
  .contexttype(ForLoop_Init)
.END();
patFinId = LLQ::DEFINE(CONTEXT)
  .contexttype(ForLoop_Finalization)
.END();
patAdjId = LLQ::DEFINE(CONTEXT)
  .contexttype(ForLoop_Adjustment)
.END();
patBodyId = LLQ::DEFINE(CONTEXT)
  .contexttype(ForLoop_Body)
.END();
patForInitIde = LLQ::DEFINE(COMPLEX)
  .query(patinitid)
  .query(patident)
.END();
patForFinIde = LLQ::DEFINE(COMPLEX)
  .query(patfinid)
  .query(patident)
.END();
patForAdjIde = LLQ::DEFINE(COMPLEX)
  .query(patadjid)
  .query(patident)
.END();
patForBodyIde = LLQ::DEFINE(COMPLEX)
  .query(patfbodyid)
  .query(patident)
.END();
iniSet = LLQ::APPLY()
  .query(patForInitIde)
patFunStat = LLQ::DEFINE(COMPLEX)
  .query(pat1)
  .query(pat2)
.END();
seto1 = LLQ::APPLY()
  .query(patFunStat)
  .on(mFunDef)
.END();

  .on(mForLoop)
.END();
finSet = LLQ::APPLY()
  .query(patForFinIde)
  .on(mForLoop)
.END();
adjSet = LLQ::APPLY()
  .query(patForAdjIde)
  .on(mForLoop)
.END();
fbodySet = LLQ::APPLY()
  .query(patForBodyIde)
  .on(mForLoop)
.END();

interPat = LLQ::DEFINE(SETOP)
  .operation(INTERSECTION)
.END();
inter1 = LLQ::APPLY()
  .query(interPat)
  .on(iniSet)
  .on(finSet)
.END();
inter2 = LLQ::APPLY()
  .query(interPat)
  .on(inter1)
  .on(adjSet)
.END();
inter3 = LLQ::APPLY()
  .query(interPat)
  .on(inter2)
  .on(fbodySet)
.END();
```

(b)

```
loopInitPat = LLQ::DEFINE(CONTEXT)
  .contexttype(ForLoop_Init)
.END();
assExpCase1 = Synthesis::
CreateASTSubTree(
  Assignment,
  assExpCase1 = Synthesis::
  CreateIdent(
    GetPrimaryIterator()
  )
);
assignInit1 =LLQ::DEFINE(SIMILARITY)
  .tree(assExpCase1)
.END();
assignsInit1 = LLQ::DEFINE(COMPLEX)
  .query(loopInitPat)
  .query(assignInit1)
.END();
initAssignments = LLQ::APPLY()
  .query(assignsInit1)
  .on(mForLoop)
.END();
```

(c)

```
loopAdjReg = LLQ::DEFINE(CONTEXT)
  .contexttype(ForLoop_Adjustment)
.END();
adjRegion = LLQ::APPLY()
  .query(loopAdjReg)
  .on(mForLoop)
.END();
loopAdjExp = LLQ::DEFINE(NODE)
  .nodetype(PostfixIncreament)
  .nodetype(PrefixIncreament)
  .nodetype(PostfixDecreament)
  .nodetype(PrefixDecreament)
  .nodetype(Assignment)
  .nodetype(AssignmentAddTo)
  .nodetype(AssignmentSubtractFrom)
  .filter(EXHAUSTIVE)
.END();
adjExp = LLQ::APPLY()
  .query(loopAdjExp)
  .on(adjRegion)
.END();
iteratorPat =LLQ::DEFINE(SIMILARITY)
  .comparison(EQUALITY)
  .tree(
    Synthesis::CreateIdent(
      *piterator
    )
  )
  .filter(EXHAUSTIVE)
.END();
adjId = LLQ::APPLY()
  .query(iteratorPat)
  .on(adjExp)
.END();
```

(d)

```
1  tmp1 = LLQ::DEFINE(CONTEXT)
2    .contexttype(ForLoop_Body)
3  .END();
4  tmp2 = LLQ::DEFINE(NODE)
5    .nodetype(ForLoop_Body)
6    .filter(DEPTHSPECIFIC_EXHAUSTIVE)
7      .depth(1)
8  .END();
```

(e)

```
9  pat4Nested4 = LLQ::DEFINE(COMPLEX)
10   .query(tmp1)
11   .query(tmp2)
12 .END();
13 mNestedForLoops = LLQ::APPLY()
14   .query(pat4Nested4)
15   .on(mForLoop)
16 .END();
```

Figure 5: CastQL code fragments for: a) the detection of level-0 loops in the function body, b) loop induction variables identification, c) identifying the induction variables' initialization expressions, d) identifying the induction variables' adjustment expressions, and e) nested loops detection.

induction variable with the code fragment of Figure 5c. An LLQ directs the search in the initialization region (lines 1-3). After that, the synthesis framework creates the prototype AST sub-tree of Figure 4b (lines 4-11). This sub-tree is used from the *assignInit1* LLQ to match any assignment expressions initializing the induction variable (lines 12-22). Furthermore, HLQForLoop identifies the loop finalization and adjustment expressions. The identification of loop adjustment expressions is shown in Figure 5d while the identification of finalization expressions is omitted since it is realized with analogous code. The search is directed into the loop adjustment context (lines 1-7) where it tries to identify expressions of the operators $i{+}{+}$, $i - -$, $+{+}i$, $- - i$, *i=...*, *i+=...*, *i-=...* (lines 8-21). It then picks the expressions in which the induction variable appears and discards the rest (lines 22-34). As a final step, HLQForLoop identifies nested loops in the current loop's body using the CastQL code fragment of Figure 5e. The *tmp1* LLQ directs the search into the loop body region and the *tmp2* LLQ uses the depth specific exhaustive filter to identify all the loops at depth 1. The resulting nested loops are assigned to the *mNestedForLoops* nodeset.

5. EXPERIMENTAL EVALUATION

The proposed framework was extensively evaluated from three different perspectives: (1) First the productivity gains achieved through the use of CastQL/FEgen are evaluated. (2) Performance related results regarding the requirements of the generated code in time and space are also discussed. (3) Finally the code generation capabilities of FEgen tool are evaluated and results regarding the size of the produced code as compared to the input grammars are provided. When code development by a human software developer is mentioned, throughout these experiments, an experienced developer with knowledge in compiler construction was used.

Figures 6a and 6b present the results of the first set of experiments where FEgen and CastQL tools are evaluated in terms of productivity. In the case of FEgen the effort consumed for developing the front ends of four languages (Figure 6a) has been determined. For each language, front ends have been developed both manually and using the FEgen tool. The average effort in man-days required for developing front ends using the FEgen tool is only 12% of the corresponding effort required for the manual development of the front ends. Taking the code size into consideration, the number of *lines of code (LOC)* produced daily has been increased by a factor of 7.5 when using the FEgen tool. CastQL has been evaluated for the development of three source code instrumentation modules that have been incorporated into MEMSCOPT source to source code optimizer [15]. These instrumentation modules are used for profiling and specifically, for memory access trace file generation, memory access metrics calculation [25], and data reuse distance measurement [26]. For each module a version based on CastQL and a manually developed C++ version have been implemented. Figure 6b compares the corresponding development efforts. Much less code has been written when using CastQL. An average reduction of LOC down to 28%, compared to manually created query engines, has been observed. This shows that CastQL improves modularity and maximizes code reuse while maintaining the performance of C++ since it is embedded to it.

For the performance evaluation of the proposed framework, regarding the requirements of the generated code in time and space, two practical usage scenarios have been tested. For each scenario, a series of code analysis and manipulation tasks used by MEMSCOPT are applied to a number of input algorithms of different sizes. The first scenario involves the application of three instrumentation modules. The second scenario includes the application of a series of loop transformations. These transformations include the expansion of all input loops' bounds by ten more iterations and the interchange of all nested loops. The codes of six test applications have been used to compose the different inputs. The applications come from the image and signal processing domain. In particular, the cavity detector [8] medical diagnostic application and five algorithms of the UTDSP benchmark suite [33] have been used. MEMSCOPT ran on a DualCore Intel Core i5 CPU at 2.50GHz with 7.85 GB of usable RAM. The size of each input is approached in terms of *thousands of LOC (KLOC)* and *total number of nodes* required for the equivalent cAST representation of the program. Figures 6c and 6d present the execution time and memory usage of the examined scenarios. Both time and space overheads are linearly related to the input LOC and the number of AST nodes. Actually, there is a slightly higher correlation to the number of AST nodes since CastQL works directly on them. Our framework's performance depends primarily on the AST size and less on other parameters such as the LOC. In any case, this difference in correlation is minimal for the examined scenarios, where the number of AST nodes has been estimated to have an average ratio of 8:1 compared to the LOC of the inputs. This ratio clearly depends on the complexity of the lines though. It may vary significantly for different coding styles and subsequently change the described correlation.

The evaluation of the automatic front end generation tool is carried out by comparing a number of statistical metrics

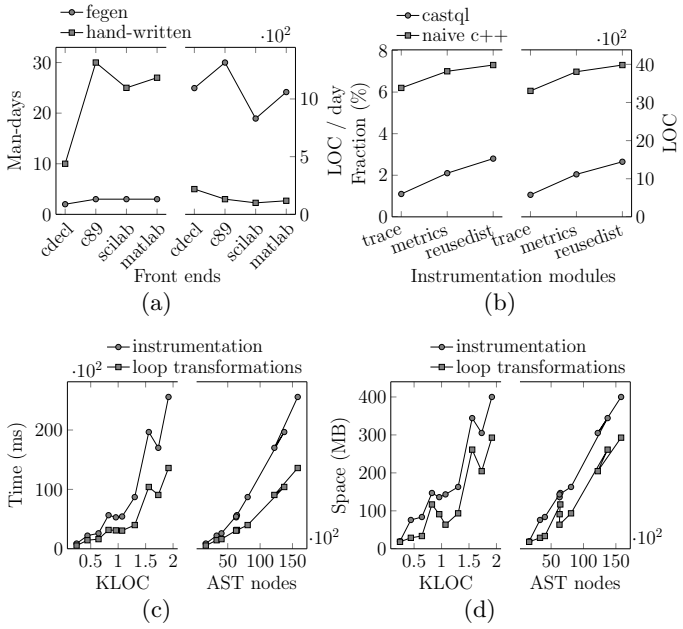


Figure 6: a) Total amount of uninterrupted labour required for the development of different front ends. b) Code size comparison between naive C++ and CastQL implementations of three instrumentation operations. The plot on the left measures the fraction of code devoted to the querying mechanism in comparison to the whole program. c) Execution time, and d) memory consumption of the two examined scenarios compared to different input sizes.

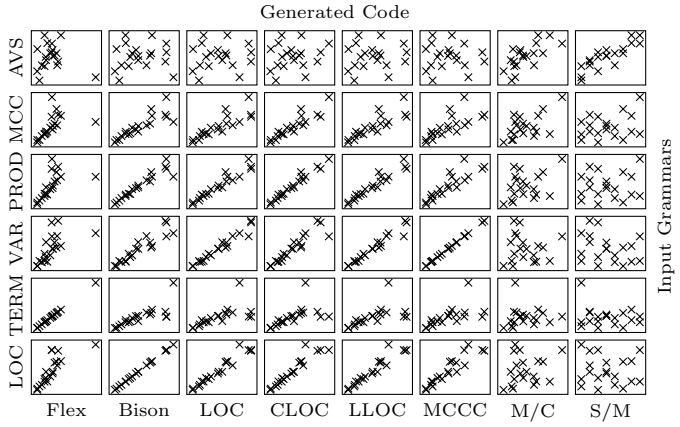


Figure 7: Correlation between the input grammars and the generated parsers. Statistical code metrics include: the number of terminal symbols (TERM), the number of nonterminals (VAR), the total number of production rules (PROD), the McCabe cyclomatic complexity (MCC) [34], the average RHS size (AVS) [34], the LOC of the generated .l file (Flex), the LOC of the generated .y file (Bison), the comment LOC (CLOC), the logical LOC (LLOC), the MCC of the generated C++ code (MCCC), the average methods/class (M/C), and the average statements/method (S/M).

regarding the size and complexity of the produced code in relation to the input languages. Twenty BNF grammars have been tested in order to prove the applicability and robustness of the generator. FEgen has been used to parse the grammars of C89, CDecl [37], Scilab, MATLAB, Ruby, PHP, C#, C++, Java, Delphi, XML, SQL, ALGOL60, Pascal, Fortran, Ada95, COBOL, BibTex, Yacc, and SML that either have been developed in the context of this research or are publicly available and have been adapted for FEgen. The generated C++ code is larger than the grammar size in terms of LOC by a factor of 11 on average. 475 lines of BNF grammar code correspond to 5220 LOC on average with no C++ coding required by the developer. In this analysis the generated lines of code concern only the process up until the parse tree generation which is completely automated by default. ASTs were specified only for the parsers of C89, Scilab, MATLAB and CDecl. An average of 30% additional LOC has been observed for these parsers compared to the ones not implementing an AST. Further insights on the comparison of the C++ code generated by FEgen and the input BNF grammars is given in the scatter plots of Figure 7. The Spearman rank correlation analysis is used to determine the correlation between the different examined metrics. As expected most of the compared pairs are highly correlated. The LOC of the input grammar and the generated Bison .y file, in particular, are extremely highly correlated since both files have essentially almost the same content (the correlation coefficient between them is 0.99). Another interesting observation is that the MCC values of the input grammars are not so highly correlated with the MCC of the generated C++ programs (0.72). This means that, the complexity of the input grammar does not have a direct impact on the complexity of the generated code. The number of nonterminal symbols on the other hand play an important role on the complexity of the code since VAR and MCCC have a correlation coefficient of 1.

6. RELATED WORK AND COMPARISONS

6.1 Comparison with Code Querying Technologies

The objective of the framework proposed in this paper is to significantly reduce the development time, effort and cost of code analysis and manipulation tools. To achieve this, concepts from both examination oriented [21, 2, 13, 12, 40, 6, 11, 3, 29, 9] and transformation oriented technologies [39, 23, 7, 22, 14, 10] are used.

Most of the transformation oriented metaprogramming environments use SDF context free grammars for the specification of the input language [39, 7, 22]. This allows them to operate on any input language. They also provide one or multiple DSLs for querying and manipulating the code. Our framework acts on any input language (one at a time) and uses a DSL to expressively query the code. The advantage of CastQL over existing work is that, since it is an internal DSL, it preserves the constructs and features of the host language. This permits the application of custom traversals written in C++, like in a classic compiler framework [31, 27], instead of using CastQL. In this way, an alternative is provided for the application of queries that cannot be expressed in the querying language. Furthermore, unlike the

aforementioned works, the tools generated by the proposed framework can be distributed as standalone C++ applications. This leverages the need for a specialized execution environment, which usually runs on a specific platform, and only a C++ compiler is required in order to use them on any platform.

On the examination oriented domain, a number of languages for querying source code and the AST exist in the literature [21, 2, 13, 12, 40, 6, 11, 3, 29, 9]. They are usually logic programming languages (Prolog-like) [21, 40, 6, 11] or relational algebra query languages (SQL-like) [12, 29]. Despite their good expressiveness and other positive aspects, most of them are limited to a specific input language in contrast to the approach followed in CastQL. Additional studies on the evaluation of code querying tools are available in [36, 1, 38].

Several approaches for code querying can also be found in the internals of many compiler infrastructures. In the simplest and most common case, they provide the ability to write custom code in order to traverse an AST representation and collect information. Roslyn [31] and Clang [27] were already mentioned in the introduction of this paper. Similar features are also available in Gecos [16], ROSE [35], Cetus [24], CIL [30], and SUIF [42]. The limited expressiveness of compiler frameworks for querying operations is their major disadvantage over code query languages. Moreover, similarly to most examination oriented systems, they only accept a predefined set of input languages.

The proposed framework combines the input language configurability of most transformation oriented systems and the querying features of examination oriented approaches for fast tools development. Furthermore, the object oriented nature of the solution and the design principles and notation patterns [18, 20] followed for the design of CastQL improve modularity and maximize code reuse as shown in the experimental evaluation section.

6.2 Automatic AST Construction

Many attempts have been made towards the automatic generation of AST representations. Wile [41] proposes an algorithm for the conversion of concrete syntax to abstract syntax. This algorithm induces the abstract syntax from an extension to the BNF notation called WBNF. Arusoaie and Vicol [4] present a generic method for inferring AST generation rules from a context free grammar. They also provide a tool which automatically generates an annotated version of a grammar using their method. Another similar method where a series of transformations are applied on the parse tree for inferring the AST is proposed in [5].

Some parser generators automate the construction of an AST through annotations in the concrete syntax. ANTLR [32] and SableCC [19] are good examples, though this option is no longer available in ANTLR as the latest version generates only a parse tree instead. Other approaches that use annotations in the concrete syntax to produce the AST include the translational BNF (TBNF) [28] and the labelled BNF (LBNF) [17] grammar notations.

The advantage of the proposed front end generator tool (FEgen) over existing methods is that it actually incorporates both of the described approaches. A recommended cAST specification is automatically deduced from the parse tree. At the same time the user can optionally define his own specification through annotations to the parse tree. This way the overhead of manually specifying an entire AST is eliminated while maintaining the ability to alter problematic parts of the automatically generated one.

7. CONCLUSIONS

In this paper, a framework for the automated development of code analysis tools in order to reduce the relevant design time, effort and costs is discussed. The proposed framework includes a number of cooperating components including a front end code generator (FEgen) and a DSL for querying the code (CastQL). A detailed discussion of these components is provided. Experimental results show that the code parsing and manipulation software generated by the proposed framework works according to grammar specification while the development time has been reduced on average down to 12% compared to equivalent manual implementations.

Future work considers the enrichment of CastQL with additional LLQ types and search strategies. The proposed framework currently misses an integrated environment that will enhance the collaboration between the individual components. A Visual Studio extension is planned to be implemented for this task in order to provide a seamless and user friendly development environment. Effort is also devoted to the practical exploitation of the proposed work. Exploitation plans of priority include the use of CastQL/FEgen in the development of a MATLAB compiler for automatic SIMD parallelization.

8. REFERENCES

[1] T. Alves, J. Hage, and P. Rademaker. A Comparative Study of Code Query Technologies. In *2011 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 145–154, Sept. 2011.

[2] P. Anderson and T. Teitelbaum. Software inspection using CodeSurfer. In *Proceedings of the 1st Workshop on Inspection in Software Engineering (WISE)*, Paris, France, July 2001.

[3] G. Antoniol, M. Di Penta, and E. Merlo. YAAB (Yet another AST browser): using OCL to navigate ASTs. In *11th IEEE International Workshop on Program Comprehension, 2003*, pages 13–22, May 2003.

[4] A. Arusoaie and D. Vicol. Automating Abstract Syntax Tree Construction for Context Free Grammars. In *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 152–159, Sept. 2012.

[5] S. Barman. *Aster: Automatic abstract syntax*. PhD thesis, University of Texas at Austin, May 2009.

[6] D. Beyer. Relational Programming with CrocoPat. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 807–810, New York, NY, USA, 2006. ACM.

[7] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1âĂŞ2):52–70, June 2008.

[8] F. Catthoor, K. Danckaert, S. Wuytack, and N. Dutt. Code transformations for data transfer and storage exploration preprocessing in multimedia processors.

IEEE Design Test of Computers, 18(3):70–82, May 2001.

[9] T. Cohen, J. Y. Gil, and I. Maman. JTL: The Java Tools Language. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 89–108, New York, NY, USA, 2006. ACM.

[10] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, Aug. 2006.

[11] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, DSL'97, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[12] O. de Moor, M. Verbaere, and E. Hajiyev. Keynote Address: .QL for Source Code Analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation, 2007. SCAM 2007*, pages 3–16, Sept. 2007.

[13] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 71–80, New York, NY, USA, 2011. ACM.

[14] C. De Roover and R. Stevens. Building development tools interactively using the EKEKO meta-programming library. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 429–433, Feb. 2014.

[15] G. Dimitroulakos, C. Lezos, and K. Masselos. MEMSCOPT: A source-to-source compiler for dynamic code analysis and loop transformations. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 385–386, Karlsruhe, Germany, Oct. 2012.

[16] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys. GeCoS: A framework for prototyping custom hardware design flows. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, volume 0, pages 100–105, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[17] M. Forsberg. *Three Tools for Language Processing: BNF Converter, Functional Morphology, and Extract*. Ph.D. Thesis, Chalmers University of Technology and Goteborg University, Goteborg, Sweden, 2007.

[18] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, Upper Saddle River, NJ, 1 edition edition, Oct. 2010.

[19] E. Gagnon and L. Hendren. SableCC, an object-oriented compiler framework. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 140–154, Aug. 1998.

[20] S. Gunther. Development of Internal Domain-specific Languages: Design Principles and Design Patterns. In *Proceedings of the 18th Conference on Pattern Languages of Programs*, PLoP '11, pages 1:1–1:25, New York, NY, USA, 2011. ACM.

[21] E. Hajiyev, M. Verbaere, and O. d. Moor. codeQuest: Scalable Source Code Queries with Datalog. In *ECOOP 2006 âĂŞ Object-Oriented Programming*, number 4067, pages 2–27. Springer Berlin Heidelberg, 2006.

[22] L. C. Kats and E. Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM.

[23] P. Klint, T. van der Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, 2009. SCAM '09*, pages 168–177, Sept. 2009.

[24] S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus âĂŞ An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Languages and Compilers for Parallel Computing*, number 2958, pages 539–553. Springer Berlin Heidelberg, 2004.

[25] C. Lezos, G. Dimitroulakos, A. Freskou, and K. Masselos. Dynamic source code analysis for memory hierarchy optimization in multimedia applications. In *Proceedings of the 2013 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 343–344, Cagliari, Italy, Oct. 2013.

[26] C. Lezos, G. Dimitroulakos, and K. Masselos. Reuse distance analysis for locality optimization in loop-dominated applications. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1237–1240, Grenoble, France, Mar. 2015.

[27] B. C. Lopes and R. Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, Aug. 2014.

[28] P. B. Mann. A Translational BNF Grammar Notation (TBNF). *SIGPLAN Not.*, 41(4):16–23, Apr. 2006.

[29] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, New York, NY, USA, 2005. ACM.

[30] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, number 2304, pages 213–228. Springer Berlin Heidelberg, 2002.

[31] K. Ng, M. Warren, P. Golde, and A. Hejlsberg. The Roslyn Project: Exposing the C# and VB compiler's code analysis. White Paper, Microsoft Corporation, Sept. 2012.

[32] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Dallas, Texas, second edition edition, Jan. 2013.

[33] S. H.-e. Peng. *UTDSP: A VLIW Programmable DSP Processor*. Master Thesis, University of Toronto, Toronto, Canada, 1999.

[34] J. F. Power and B. A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, Nov. 2004.

[35] D. Quinlan. Rose: compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, June 2000.

[36] P. Rademaker. *Binary relational querying for structural source code analysis*. Master Thesis, Universiteit van Utrecht, Utrecht, the Netherlands, 2008.

[37] T. Stripf, O. Oey, T. Bruckschloegl, J. Becker, G. Rauwerda, K. Sunesen, G. Goulas, P. Alefragis, N. S. Voros, S. Derrien, O. Sentieys, N. Kavvadias, G. Dimitroulakos, K. Masselos, D. Kritharidis, N. Mitas, and T. Perschke. Compiling Scilab to high performance embedded multicore systems. *Microprocessors and Microsystems*, 37(8, Part C):1033–1049, Nov. 2013.

[38] R.-G. Urma and A. Mycroft. Programming Language Evolution via Source Code Query Languages. Tucson, AZ, USA, Oct. 2012.

[39] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. d. Jong, M. d. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-environment: A Component-Based Language Development Environment. In *Compiler Construction*, number 2027, pages 365–370. Springer Berlin Heidelberg, 2001.

[40] K. D. Volder. JQuery: A Generic Code Browser with a Declarative Configuration Language. In *Practical Aspects of Declarative Languages*, number 3819, pages 88–102. Springer Berlin Heidelberg, 2005.

[41] D. S. Wile. Abstract Syntax from Concrete Syntax. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 472–480, New York, NY, USA, 1997. ACM.

[42] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Not.*, 29(12):31–37, Dec. 1994.