Towards a generic framework for analyzing C++ in Rascal

Rodin T.A. Aarssen
CWI Amsterdam, and
Eindhoven University of Technology
The Netherlands
Rodin.Aarssen@cwi.nl

Jurgen J. Vinju
CWI Amsterdam, and
Eindhoven University of Technology
The Netherlands
Jurgen.Vinju@cwi.nl

***Abstract*—Maintenance of legacy software is often a costly task. Software analysis tools can help a lot by providing insight in existing code. However, for C++, this tooling often doesn't give satisfactory answers. In this extended abstract, we introduce ClaiR, our generic C++ analysis framework we built on top of the meta-programming language Rascal.**

***Index Terms*—Reverse engineering, Rascal, C++**

I. Introduction

A legacy software system is a piece of software that has had a significant lifetime. This means that it has been developed for a long time, by many people, and with programming paradigms changing over time. In practice, this generally leads to software systems that are unnecessarily complex, and therefore hard and expensive to maintain. In industry, it is often not an option to start over with a fresh system, as a lot of implicit knowledge and business logic is encoded in the legacy system.

This paper presents our ongoing work towards an analysis framework for C++ in Rascal.

A. Motivation

We recently started the MERITS (Model Extraction for Re-engIneering Traditional Software) project, in which we investigate how we can compare and combine software models created using active learning with models created using static analysis. In the project, our focus is on the extraction of the latter type of models. In order to obtain these models, we need a more abstract view on software than the code itself. To this end, we created ClaiR (C language analysis in Rascal)[1] [1], which serves as a front-end plugin for the meta-programming language Rascal. Combining ClaiR and Rascal, we can create abstract syntax trees (ASTs) from source code, which serve as a starting point for language analysis and model extraction.

We chose to use Rascal [7] as an analysis platform, because it allows us to generalize the project's problem setting and lay a foundation for C++ analysis in general. Furthermore, it allows us to reuse Rascal functionality that has proven to be effective for program analysis of e.g. Java and PHP code [6], [8].

One of the main goals of ClaiR is to provide better IDE support for C++, for instance by providing visualization, and performing static analysis on source code. Also, we want to create a framework in which software engineers can easily write their own code analysis tools.

B. Requirements

The C++ language comes in many variants. ClaiR has been designed not to be specifically aimed towards one of those flavors. In particular, we want to provide support for the extensions that have been added to the language in the Microsoft Visual C++ compiler, as this compiler is widely used in industry.

One of the goals of ClaiR is that its representation of the code still resembles the original code; in this way, the ASTs created by ClaiR are an intuitive abstraction of the code. Additionally, we require that manipulation of and data extraction from the ASTs can be performed in an intuitive – mathematical – way.

C. Types of analyses

ClaiR's ASTs are the starting point for code analysis; depending on the question, many views on the code could be generated, such as call graphs, control flow graphs, and data flow graphs. Using and combining these views can then provide an answer to the question.

II. Architecture and features

Rascal [7] is a meta-programming language, aimed at software analysis and transformation. It is a functional language – yet syntactically, it supports many imperative constructs – based on term rewriting and relational calculus primitives. In ClaiR, we defined an abstract data type to represent C++ code in Rascal. At the time of writing, it consists of 339 constructors over 10 data types. For instance, the constructor `multiply(Expression lhs, Expression rhs)` is the abstract representation of a multiplication in the source.

ClaiR does not include its own C++ parser; instead, it reuses Eclipse's C++ Development Tooling (CDT)[2] to parse source code. We chose CDT as our external parser, because it has an abstract representation that still resembles the original source

[1] https://github.com/cwi-swat/clair

[2] https://www.eclipse.org/cdt/

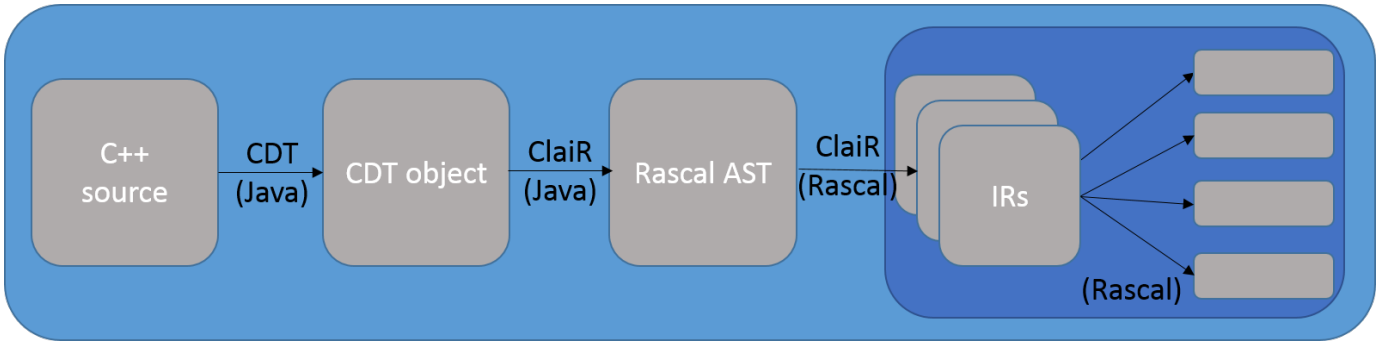

Fig. 1: The architecture of ClaiR. The grey boxes represent the involved artifacts. The connecting arrows are labeled with the name of the software that handles the transformation, and the implementation language. At the time of writing, the parts in the dark blue box have not yet been (fully) implemented.

code. This is in contrast with compiler front-ends, which are more oriented towards a lower-level internal representation. CDT has been used for code refactorings [5], which gives us confidence that it is suitable for other analyses as well: for refactoring, a detailed view of the code is necessary to perform semantics-preserving code transformations.

We will now elaborate on ClaiR's architecture, which is depicted schematically in Fig. 1.

This process yields a Java object containing, among other things, CDT's abstract view of the source code under analysis. Then, ClaiR's adapter takes this Java object and transforms it to a Rascal object. More specifically, it returns an AST, which is an instance of the data model we described earlier. This representation is the basis for further analysis.

There are various reusable intermediate representations possible at this point. For instance, one could extract a call graph or a control flow graph [3]. Combining several of these intermediate results allows the user to answer questions about the code. Currently, call graph extraction is implemented in ClaiR.

A. *Features*

This section summarizes the key features of ClaiR.

Name resolution ClaiR performs name resolution for every name encountered in the source code. This means that every use of a name can be linked to its declaration, and vice versa.

Type resolution For every expression, ClaiR resolves the type of the expression.

Traceability Every node in a ClaiR AST has an attribute containing the original source location it corresponds with.

Include resolving If possible, ClaiR resolves the include directives that are present in the source code. This is instrumental for correct name resolution.

Macros ClaiR handles macros in their expanded form. This ensures that the syntax of the code under analysis is not invalidated by the use of macros, e.g. by hiding curly braces under a macro definition.

Visualization ClaiR has some IDE-integrated visualization features. By selecting a piece of code in an Eclipse editor, a user can generate the corresponding AST in text, or generate a visual representation of that AST.

Pattern matching The underlying platform Rascal provides strong pattern matching functionality. Because ClaiR's ASTs are Rascal objects, this functionality can also be used on ClaiR ASTs. Pattern matching is a very powerful technique for e.g. detection of design patterns and code smells.

Relational algebra Rascal contains the relational model M^3, which is a general model for code analysis in Rascal [4]. An M^3 model contains several relations that describe certain properties of the source code; using relational algebra, these relations can be combined to extract data from the model. ClaiR implements the M^3 model for C++.

Not dialect-specific We want ClaiR to be a generic framework for analyzing C++. Therefore, we do not want it to be tied to a specific dialect of C++. Specifically, we want ClaiR to support Microsoft-specific additions to the language. For this, we have adapted CDT's parser to add support for these constructs.

III. Demonstration

For a demonstration of ClaiR's capabilities, let's consider the piece of C++ code from Listing 1, which calculates the factorial of a positive integer. In Listing 2, the AST as produced by ClaiR's parsing and adapting is given. To preserve space, the type resolution and physical source location attributes are omitted. The AST does show the results of name resolution: every occurrence of the `name` constructor comes with a `decl` attribute. This attribute, with a URI-like syntax, is the internal representation of the corresponding name. An example of ClaiR's visualization features is shown in Fig. 2, which is a visual encoding of the return argument of the code from Listing 1.

```
int factorial(int n) {
    return n<=1 ? 1 : n * factorial(n-1);
}
```

Listing 1: Example C++ code.

```
functionDefinition(
  declSpecifier(
    integer()),
  functionDeclarator(
    name("factorial"),
    [parameter(
        declSpecifier(
          integer()),
        declarator(
          name("n"),
          decl=|cpp+parameter:///factorial(int)/n|))],
    decl=|cpp+function:///factorial(int)|),
  compoundStatement(
    [return(
        conditional(
          lessEqual(
            idExpression(
              name("n"),
              decl=|cpp+parameter:///factorial(int)/n|),
            integerConstant("1")),
          integerConstant("1"),
          multiply(
            idExpression(
              name("n"),
              decl=|cpp+parameter:///factorial(int)/n|),
            functionCall(
              idExpression(
                name("factorial"),
                decl=|cpp+function:///factorial(int)|),
              [minus(
                  idExpression(
                    name("n"),
                    decl=|cpp+parameter:///factorial(int)/n|),
                  integerConstant("1"))])))])))
```

Listing 2: ClaiR AST of the code from Listing 1.

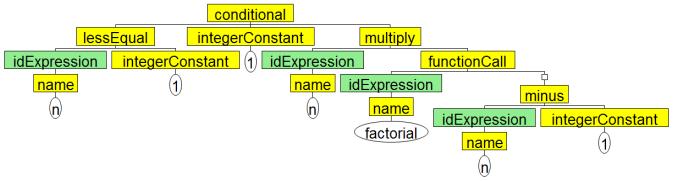

Fig. 2: Visualization of part of the AST from Listing 2.

IV. Conclusion

Code analysis tooling is essential in modern software engineering. In this extended abstract, we have introduced ClaiR, which lays the foundation for a static analysis framework for C++. With ClaiR, we have created a platform on which software engineers can create their own code analysis tools for C++. ClaiR supports pattern matching on its ASTs, and supports code analysis with relational algebra.

A. Future work

ClaiR's ASTs are lossy in the sense that the whitespace and comments from the original source code are not represented. We aim to add support for lossless AST creation in the future, to make it possible to fully regenerate the original source code from ClaiR's syntax trees. Likewise, we aim to encode whether or not an AST node originates from a macro expansion.

Currently, ClaiR can deliver an M^3 model and a call graph from source code. We aim to provide more analysis algorithms for ClaiR, and specifically, we will work towards extracting actions models from code.

At the moment, pattern matching on abstract patterns is a rather tedious procedure, as it requires the programmer to exactly type in the patterns. As (nested) patterns can grow in size fairly quickly, this is an error-prone task. Therefore, we want to provide pattern matching on concrete syntax [2], [9], [10]. In this situation, a user can provide actual C++ code as a "pattern", and match that with an AST. This abstracts away from the abstract representation, and will make pattern matching more accessible. As a corollary, implementing this in Rascal will make concrete pattern matching available for all languages in Rascal using an external parser.

References

[1] R. Aarssen, "C language analysis in Rascal (ClaiR), Rascal plugin," Sep. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.891122

[2] A. Aasa, K. Petersson, and D. Synek, "Concrete syntax for data objects in functional languages," in *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM Press, 1988, pp. 96–105.

[3] A. Aho, R. Sethi, and J. Ullman, *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju, "M3: A general model for code analytics in Rascal," in *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, March 2015, pp. 25–28.

[5] E. Graf, G. Zgraggen, and P. Sommerlad, "Refactoring support for the C++ development tooling," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 781–782. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297885

[6] M. Hills, P. Klint, and J. Vinju, "An empirical study of PHP feature usage: A static analysis perspective," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 325–335. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483786

[7] P. Klint, T. van der Storm, and J. Vinju, "Easy meta-programming with rascal," in *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, ser. GTTSE'09. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 222–289. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949925.1949932

[8] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of Java reflection: Literature review and empirical study," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 507–518. [Online]. Available: https://doi.org/10.1109/ICSE.2017.53

[9] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *LISP 1.5 Programmer's Manual*. Cambridge, Mass.: The MIT Press, 1966.

[10] M. Sellink and C. Verhoef, "Native patterns," in *Proceedings of the Fifth Working Conference on Reverse Engineering*, M. Blaha, A. Quilici, and C. Verhoef, Eds. IEEE Computer Society Press, 1998, pp. 89–103.