Electronic Communications of the EASST
Volume X (2015)

Proceedings of the Third International Workshop on Open and Original Problems in Software Language Engineering (OOPSLE 2015)

Domain-Specific Languages for Program Analysis

Mark Hills

4 pages

Guest Editors: Anya Helene Bagge, Vadim Zaytsev
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Domain-Specific Languages for Program Analysis

Mark Hills[1]

[1] mhills@cs.ecu.edu, http://www.cs.ecu.edu/hillsma
Department of Computer Science
East Carolina University, Greenville, NC, USA

Abstract: Program analysis is an important aspect of many software language engineering tools. As part of our work on creating program analysis tools and frameworks in Rascal, we are looking at creating domain-specific languages to support different program analysis tasks, reducing the effort to develop new analysis tools and providing effective notations and libraries. We have created a language for defining control flow for the purpose of building control flow graphs, and are looking at languages for other analysis tasks, such as generating function summaries for library functions based on program documentation.

Keywords: program analysis, domain-specific languages, language frameworks

1 Introduction

Program analysis is an important part of building useful software language engineering tools. While, in some cases, this is restricted to just name resolution and type checking, often more sophisticated analyses are needed. Examples include type inference, call graph construction, alias analysis, various forms of security analysis (e.g., taint analysis), and maybe even resource analysis for languages where estimating and possibly bounding resource consumption is critical.

For many languages—especially custom domain-specific languages—these program analysis tools need to be constructed from scratch. Often these tools have many similarities to tools for other languages, with significant amounts of boilerplate mixed in with language-specific functionality. As part of our work on creating program analysis tools and frameworks in Rascal, we are also looking at creating domain-specific languages for different program analysis tasks, allowing us to define the language-specific functionality needed for the analysis while providing support for commonly-recurring patterns through code generation and libraries.

The rest of this paper is organized as follows. Since the context of our work is the Rascal language, Section 2 provides a brief introduction to Rascal. Section 3 then describes our ongoing work on domain-specific languages for program analysis, including an existing language for control-flow graph generation as well as languages for other program analysis concerns that are currently being designed. Section 4 then briefly discusses related work, while Section 5 poses some questions for discussion.

2 Rascal

Rascal [7, 8] was designed to be a "one-stop shop" for the domain of meta-programming, with one of the main use cases being program analysis. To support this, the language includes a number

```
rule EXP::add = entry(left) --> right --> exit(self);
rule EXP::sub = ^left --> right --> $self;
rule STATEMENT::whileStat =  create(footer),
                             ^exp -conditionTrue-> body -backedge-> exp,
                             exp -conditionFalse-> $footer;
```

Figure 1: Several DCFLOW Rules for Pico.

of features well-suited to this domain, including: extensive support for manipulating concrete and abstract program representations; a rich set of data types, including locations—URIs for representing source code locations or external sources of data, usable as Rascal resources [4]—and relations with relational algebra operations; flexible pattern matching; and built-in support for traversals and fix-point computation. Further information, including online documentation and lecture slides, is provided online at http://www.rascal-mpl.org.

3 DSLs for Program Analysis

Although Rascal provides support for building a wide variety of program analysis tools, it does not include any special support for common analysis tasks, such as control-flow graph extraction, name resolution, type inference, or extraction of analysis information from the documentation of existing libraries. Because of this, we have begun to look at libraries and small domain-specific languages tailored to these tasks.

3.1 A DSL for Control-Flow Graph Construction

DCFLOW [3] is a declarative, domain-specific language, with supporting libraries, for defining the control flow rules for a programming language. A fragment of an existing DCFLOW script, showing examples of several rules and DCFLOW language features, is shown in Figure 1. Rules in DCFLOW are given over the abstract syntax of a programming language, represented by Rascal algebraic data types (ADTs). The first rule, for the `add` expression, states that control enters at the left operand (named `left` in the ADT), then goes to the right operand (similarly named `right`), then exits from `self`, representing the entire expression. This models our understanding of an addition expression: we would first evaluate the left operand to find its numeric value, followed by the right operand, then we actually perform the addition. The second rule defines the same control flow for the `sub` expression, but uses syntactic sugar, with `^` marking the entry and `$` marking the exit. The third rule shows the control flow for a `while` statement: control starts at the condition and, when this is true, goes to the body, and then back to the condition. Alternatively, if the condition is false, control goes to a footer node, synthesized with `create(footer)`. This footer also acts as the exit from the construct.

Once a DCFLOW specification is created, the DCFLOW translator converts the specification into a collection of Rascal modules. These modules handle the labeling of the AST, which assigns unique IDs to each instruction, and the creation of a control flow graph for an input program, based on the DCFLOW rules specifying the control flow for the input language. The CFG construction

process uses the generated modules, language-specific functions provided by the user,[1] and DCFLOW libraries to actually perform the control flow graph generation, creating one or more control flow graphs for an input program. These graphs can be used in analysis algorithms, and can also be visualized using DCFLOW visualization functionality, which generates GraphViz diagrams using the dot language.

3.2 Program Tracing and Summary Extraction

Along with continuing work on DCFLOW, we are also planning additional DSLs for common tasks such as working with program traces and extracting analysis information from program documentation. For the first, we are designing an internal DSL to filter gathered execution traces and register functionality that can trigger to handle interesting events. This is initially being designed to support dynamic analysis of PHP code, but is intended to be flexible enough to provide similar support for program traces gathered from other languages.

Our current work on summary extraction is focused on designing a template- based language for extracting information from online documentation of program libraries. The initial goal is to extract function summary information for PHP libraries related to types, with the ability to attach additional summary information for use by other analyses. This has already been done for PHP using HTML scraping, but we want to create a more robust solution that can easily be updated, expanded to more sources of documentation, and queried without needing to be fully loaded into memory (e.g., by using a document database like MongoDB).

4 Related Work

The DEFACTO system [1] uses RScript [6], which supports n-ary relations and relational algebra, as a query language for extracting and forming relations over analysis facts, as does Vankov's work on formulating program slicing using relational techniques [11]. Rascal [7, 8], the language we use in this paper, has n-ary relations as a native datatype; relational operations, such as transitive closure, are built in to the language. JastAdd [2] provides support for the declarative definition of control flow rules and dataflow analysis algorithms based on abstract syntax trees [10], similar to the DCFLOW language discussed above. Other work on using focused domain-specific languages to support program analysis tasks includes the DHAL language [9] and its variants for data flow analysis, and an approach for performing incremental name and type analysis [12] implemented as part of the Spoofax language workbench [5].

5 Discussion

We believe this work raises a number of interesting discussion questions for the workshop, including the following questions:

- In which cases does it make sense to create a new (internal or external) DSL versus just directly using a language like Rascal?

[1] These allow Rascal code to be written to handle functionality that cannot be handled directly in DCFLOW.

- Which kind of DSL (internal or external) is best suited to which kinds of problems? When should we support both (e.g., by translating an external DSL into uses of a fluent interface)?
- How can we best add support for complex features without tailoring the DSL too closely to a specific implementation of these features?

Bibliography

[1] H. J. S. Basten and P. Klint. DeFacto: Language-Parametric Fact Extraction from Source Code. In *Proceedings of SLE'08*, volume 5452 of *LNCS*, pages 265–284. Springer, 2008.

[2] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of OOPSLA'07*, pages 1–18. ACM Press, 2007.

[3] M. Hills. Streamlining Control Flow Graph Construction with DCFlow. In *Proceedings of SLE 2014*, volume 8706 of *LNCS*, pages 322–341. Springer, 2014.

[4] M. Hills, P. Klint, and J. J. Vinju. Meta-language Support for Type-Safe Access to External Resources. In *Proceedings of SLE'12*, volume 7745 of *LNCS*, pages 372–391. Springer, 2012.

[5] L. C. L. Kats and E. Visser. The Spoofax Language Workbench. In *OOPSLA 2010 Companion*, pages 237–238. ACM, 2010.

[6] P. Klint. Using Rscript for Software Analysis. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.

[7] P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *Post-Proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[8] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM'09*, pages 168–177. IEEE, 2009.

[9] L. Moonen. Data Flow Analysis for Reverse Engineering. Master's thesis, University of Amsterdam, 1996.

[10] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 78(10):1809–1827, 2013.

[11] I. Vankov. Relational approach to program slicing. Master's thesis, University of Amsterdam, 2005.

[12] G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A Language Independent Task Engine for Incremental Name and Type Analysis. In *Proceedings of SLE'2013*, volume 8225 of *LNCS*, pages 260–280. Springer, 2013.