

Scripting Refactorings with JunGL

Mathieu Verbaere Arnaud Payement Oege de Moor

Programming Tools Group
University of Oxford
United Kingdom

{Mathieu.Verbaere, Arnaud.Payement, Oege.de.Moor}@comlab.ox.ac.uk

Abstract

We describe *JunGL*, a language to script refactoring transformations. It manipulates a graph representation of the program, including extensible semantic information such as variable binding and dataflow. JunGL enables the full automation of complex refactorings: finding program elements of interest, checking preconditions and performing the transformation itself.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.3.4 [Programming Languages]: Processors; D.2.6 [Software Engineering]: Programming Environments

General Terms Languages, Design

Keywords Refactoring, scripting language, source code transformation, language workbenches

1. Introduction

A *refactoring* is a program transformation that improves the design of a program while preserving its behaviour [7]. Often the purpose is to correct existing design flaws, to prepare a program for the introduction of new functionality, or to take advantage of a new programming language feature such as generic types. Most of these transformations require tedious, error-prone manipulation of the source code, and it is therefore desirable to provide automated support for applying them.

Mainstream IDEs now provide such support, but as a fixed menu of refactorings for instance for renaming, extracting a method, extracting an interface, and so on. They do not provide a way to express customized source code transformations, that developers may wish to perform, since conventional search-and-replace mechanisms, even structural one, are not powerful enough to express complex refactorings like the ones cited above. Moreover, their support of refactoring sometimes contains bugs [11].

This is due to the complexity of correctly implementing refactorings: it requires the same kind of analysis as in compiler optimisations, but at the source level. A framework for refactoring must provide dataflow analysis facilities as well as other, perhaps more obvious, features such as pattern matching and mechanisms for variable binding.

A wealth of work has been done, over the past fifteen years, in the formal specifications of compiler optimisations, and in generating program transformers from such specifications, *e.g.* [1, 4, 5, 13]. All these works contrast with research that seeks to express transformations in purely syntactic terms, without initial emphasis on dataflow analysis, such as [2] or in term rewrite systems where the specification of dataflow facts [6] is much less declarative than in JunGL. The situation is reversed, however, when specifying the actual changes to the object program — currently, these are very operational in JunGL.

The contributions of the work presented here are:

- the identification of the need for a language to script refactorings,
- the integration of all the required features in a clean and coherent design,
- the validation of that design through several non-trivial examples,
- an implementation of the language on the .NET platform.

2. JunGL

JunGL – short for *Jungle Graph Language* – is a hybrid of a functional language in the tradition of ML and a logic query language akin to Datalog (essentially Prolog without data structures). Like ML, it has features such as pattern matching, higher-order functions, and allows the use of updatable references to manipulate the graph representation of the program. In addition, we introduce the notion of *predicates* for users to run a number of queries and find out specific information. Indeed, JunGL supports Datalog queries to express complex relationships between program elements, and *path queries* as a convenient shorthand. Path queries are special predicates (written like regular expressions) that identify paths in the program graph. They can capture, for instance, hierarchical information or dataflow properties in a very neat syntax.

Before manipulating an object program, the structure of the syntax tree must be specified within the scripts by declaring a hierarchy of new types of nodes, possibly with pretty printing annotation. For example:

```
type Expression =
  | @pretty("$name") Var = { name: string }
  | @pretty("$value") Int = { value: int }
  | @pretty("' (' $expression ') '")
    ParenthesizedExpression = { expression: Expression }
```

Currently, the system does not handle parsing itself, and hence parsers that build the initial raw tree from given source files or libraries have to be provided.

Additional computed information is then added via *lazy* edges definitions, which will only be evaluated when their value is required. To illustrate,

```
let edge use x:Expression → ?y = [x]child*;lookup[?y]
```

defines potential edges that relate an expression to the variables it refers to. This reads as a path query: y is the declaration of a variable used in the expression x if we can reach it from the node x by traversing zero or more *child* edges followed by a *lookup* edge. The child edge is a primitive in JunGL, whereas the lookup one, which relates a variable reference to its declaration, is defined in JunGL using similar path queries. Of course its complexity depends on the object language. For a substantial subset of $C^\#$ (including generics), the declaration and type lookup edges are built from about fifty sub-edges and predicates, by naturally translating into JunGL the ECMA specifications of the language [9]. That mechanism of lazy edges construction is a major feature of JunGL: it is very convenient when introducing new graph nodes, as all the semantic information is automatically constructed and maintained.

Finally, in a transformation script, one can use predicates in functions via a stream comprehension, to find program elements with a particular property. Hence,

```
let predicate p(?x) = ... in
{ ?x | p(?x) }
```

will return a stream of all x that satisfy the predicate p .

3. Refactoring examples

To assess the design of JunGL, we have implemented three of the most frequently used refactorings for a non-trivial subset of the $C^\#$ language. More information is available in [8, 11].

Rename Variable Automatically renaming a variable requires variable binding information and the ability to detect potential conflicting declarations of variables with a similar name. We use path queries to check whether a variable reference is in the scope of a variable definition with a similar name. Whenever variable hiding occurs, we try not to reject the transformation and instead remove the ambiguity by adding, for instance, the explicit ‘this’ qualifier to any reference to a newly hidden instance member.

Extract Method There are four phases: checking validity of the selection, determining what parameters must be passed, where declarations should be moved, and finally doing the transformation itself. The two first steps require control and dataflow information. For instance, in order to qualify a variable x as a value parameter, we need to check that its value is not live at the end of the selection we wish to extract. In JunGL, this dataflow fact about x can be expressed with the predicate

```
! [endNode](local ?z: cfsucc[?z] & ![?z]def[x])+[?u]use[x]
```

which requires the absence, in the control flow graph, of a path from the end of the selection to a potential use of x , where no intermediate node z defines x .

Extract Interface This type-based refactoring alters the type structure of a program [10]. It merely consists in collecting some type constraints over the original program and trying to solve them to find out whether some variables can be given the type of the newly introduced interface. Our constraint solver is currently external to JunGL, but we plan to integrate it soon, since type-based refactorings are very common.

4. Implementation

JunGL is implemented on the .NET platform using $C^\#$ and $F^\#$. It consists of three components: the graph data structure, the inter-

preter for the scripts, and an editor that makes the development of the scripts very interactive by allowing the visualization of freshly defined edges. Interestingly, JunGL does not require the full power of a logic language such as Prolog. Datalog provides just the right balance of expressive power (path queries can easily be translated to Datalog) with an efficient implementation.

5. Conclusion

We have designed a language that allows quick and concise implementation of refactoring transformations. That design was validated through three refactorings, very different in nature, for a substantial subset of $C^\#$. We are now confident that JunGL is expressive enough to cope with many issues arising in source code transformations. A missing feature, however, is the ability to use concrete syntax for object programs [12]. Also we hope to make JunGL scale to industrial-size projects, using CodeQuest [3] as a backend to evaluate Datalog queries.

Acknowledgments

We thank Microsoft Research for their generous support of this project.

References

- [1] Uwe Alßmann. OPTIMIX — a tool for rewriting and optimizing programs. In *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools*, pages 307–318. 1998.
- [2] James R. Cordy. TXL - a language for programming language tools and applications. In *International Workshop on Language Descriptions, Tools and Applications*, Electronic Notes in Theoretical Computer Science, pages 1–27, 2004.
- [3] Elnar Hajiyeve, Mathieu Verbaere, and Oege de Moor. CodeQuest: scalable source code queries with Datalog. In *Proceedings of ECOOP*, 2006.
- [4] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow and analyses via local rules. In *Proceedings of the 32nd ACM symposium on Principles of Programming Languages*, pages 364–377, 2005.
- [5] Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-order and symbolic computation*, 16(1-2):15–35, 2003.
- [6] Karina Olmos and Eelco Visser. Strategies for source-to-source constant propagation. In *Workshop on Reduction Strategies in Rewriting and Programming*, Electronic Notes in Theoretical Computer Science, 2002.
- [7] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [8] Arnaud Payement. Type-based refactoring using jungl. MSc thesis, University of Oxford, 2006.
- [9] C# Language Specification. Standard ECMA-334, 2006.
- [10] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Proceedings of OOPSLA*, pages 13–26, 2003.
- [11] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In *Proceedings of ICSE*, 2006.
- [12] Eelco Visser. Meta-programming with concrete object syntax. In *Generative programming and component engineering*, pages 299–315, 2002.
- [13] Deborah Whitfield and Mary Lou Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.