Generic Software Transformations

Jan Heering[1] and Ralf Lämmel[2]
[1] CWI, Amsterdam, Jan.Heering@cwi.nl
[2] CWI and Vrije Universiteit, Amsterdam, ralf@cwi.nl

A generic software transformation is a *transformation scheme* that can be instantiated to an actual transformation by supplying a language and a language concept as arguments. For instance, a generic **extract** might be instantiated to

- **extract**[C, variable] (common subexpression elimination in C);
- **extract**[EBNF, non-terminal] (elimination of common parts of right-hand sides of EBNF syntax rules);
- **extract**[C, function] (function folding in C);
- **extract**[Java, method] (method folding in Java);
- **extract**[C++, template] (folding class definitions into template instances in C++).

Generic transformations capture the common principles underlying transformations across different languages as well as different constructs for the same language. The languages involved are not limited to general purpose programming languages, but also include domain-specific languages (like EBNF) and modeling languages (like UML).

Generic transformation primitives Some categories of language constructs, such as abstractions and applications, immediately suggest primitives for building generic transformations. In fact, the extraction illustrated above can be broken down in a sequence of primitive transformations for *introduction* and *folding* [2]. Introduction amounts to insertion of a new abstraction (for instance, a method) whose body is equivalent to the fragment of interest without creating any conflicts. Folding replaces the fragment of interest by the application of the abstraction (for instance, a method call). Their inverses, elimination and unfolding, are themselves again primitives.

Crosscutting language concepts like scope and typing suggest further primitives. A simple one for typing is adding a type annotation for languages with type inference. That is, an inferred type is made explicit in the source code. Scoping primitives affect the scope of bindings or the way entities are brought into a scope. A simple example is moving a local declaration to the top level so that it becomes global. Another example is replacement of a hard-wired reference to a declaration by a reference to a newly introduced parameter.

Other primitives concern more specific constructs or concepts that are nevertheless part of many languages. These capture conditional algebraic laws for the introduction and elimination of the constructs in question. For instance, subtype polymorphism is found in many object-oriented languages. There is a correspondence between cascaded conditionals and polymorphic dispatch, which leads to transformations for their

elimination and introduction. Another example concerns regular expression operators, which are found in many languages. Again, the elimination and introduction of regular operators can be described in a generic way, while EBNF to BNF and BNF to EBNF conversions are specific instantiations of this idea.

Generic code smells The notion of *code smell* was introduced in the context of object-oriented software refactoring, but a generic notion that is parameterized with a language and a *criterion* or *metric* is applicable to the much wider class of languages outlined above as well as to other types of transformations. For object-oriented languages and static or structural criteria, (a special case of) the original notion is obtained. Other criteria may be related to security or performance. Such smells indicate parts of the code that may benefit from security enhancing or optimizing transformations. Some possible criteria and corresponding smells are:

- distance, e.g., too many levels of blocks between declaration and use,
- coupling (in the sense of object-oriented programming),
- size, e.g., methods that are too long (in terms of lines of code),
- style, e.g., missed opportunities for using higher-level idioms,
- extensibility, e.g., fragile use of hardwired conditionals,
- generality, e.g., fixed variation points including overly specific types,
- readability, e.g., too concise use of higher-order functions.

Generic transformation frameworks Current work on semantics is yielding techniques for the modular definition of language concepts. A generic transformation framework is supposed to support a similar degree of modularity. In fact, the underlying ontology of such a transformation framework has to be aligned to the ontology of language concepts. More than this, we seek strictly aligned definitions of semantics and transformations. That is, a complete framework for language concepts must provide all of the following: syntax, static analysis, execution semantics (if any), primitive transformations, and possibly other ingredients related to the use of transformations in actual scenarios. In [1] it is argued that *equational logic* can serve to unify the definition of static analysis (as abstract interpretation), execution semantics, and program transformation. The benefits of such a unified view are considerable. In particular, equational specifications of interpreters, transformation systems, or analyzers can often be done in a modular fashion by adding or removing sets of equations.

References

[1] J. Field, J. Heering, and T. B. Dinesh. Equations as a uniform framework for partial evaluation and abstract interpretation. *ACM Computing Surveys*, 30(3es), September 1998. Electronic supplement: 1998 Symposium on Partial Evaluation.

[2] R. Lämmel. Towards generic refactoring. In *Proc. 2002 ACM SIGPLAN Workshop on Rule-Based Programming (RULE '02)*, pages 15–28. ACM Press, 2002.