Code Analyses for Refactoring by Source Code Patterns and Logical Queries

Daniel Speicher, Malte Appeltauer, Günter Kniesel
Dept. of Computer Science III, University of Bonn - Germany
{dsp, appeltauer, gk}@cs.uni-bonn.de - roots.iai.uni-bonn.de

***Abstract*—Preconditions of refactorings often comprise complex analyses that require a solid formal basis. The bigger the gap between the level of abstraction of the formalism and the actual implementation is, the harder the coding and maintenance of the analysis will be. In this paper we describe a subset of GenTL, a *gen*eric analysis and *t*ransformation *l*anguage. It balances the need for expressiveness, high abstractness and ease of use by combining the full power of a logic language with easily accessible definitions of source code patterns. We demonstrate the advantages of GenTL by implementing the analyses developed by Tip, Kiezun and Bäumer for generalizing type constraints [11]. Our implementation needs just a few lines of code that can be directly traced back to the formal rules.**

I. Introduction

Preconditions for refactorings require thorough analyses based on a solid formal foundation. In order to reduce the implementation effort, risk of errors, and cost of evolution it is desirable to have an implementation language at a similar abstraction level as the formal foundation. Part of this problem has been addressed by various approaches to logic meta-programming [4], [13]. They have demonstrated that the expressive power of logic meta-programming enables the implementation of powerful program analyses for refactoring (e.g. [12]).

Unfortunately, logic meta-programming requires programmers to know the meta-level representation of the analysed language and to think and express their analyses in terms of this representation. For instance, JTransformer, our own logic meta-programming tool for Java [8], [5] represents methods by a predicate with seven parameters. The full Java 1.4 AST is represented by more than 40 predicates. Mastering these predicates and the precise meaning of each of their parameters can be error-prone and forces programmers to think at the abstraction level of the meta-representation.

In this paper we offer the expressive power of logic meta-programming but raise the abstraction level by providing means of expressing constraints on the structure of source code elements without having to learn a new API. This is achieved by a predicate that selects source elements based on source code patterns containing meta-variables as place-holders. Thus the concept of meta-variables is all that programmers have to learn in addition to mastering the analysed language. The corresponding concepts of GenTL are introduced in Section II. In Section III we introduce the problem of type generalizing refactorings and the corresponding formal basis elaborated by Tip, Kiezun and Bäumer in [11]. In Section IV we show how GenTL can easily express the analyses of [11].

II. GenTL

GenTL is a generic program analysis and transformation language based on logic and source code patterns. For lack of space we describe here only its analysis features. We start by the introduction of meta-variables and code patterns, then we introduce the selection of program elements based on code patterns and finally we show how arbitrary predicates can be easily built on this simple infrastructure.

A. Code Patterns

A *code pattern* is a snippet of base language code that may contain meta-variables. A *meta-variable* (MV) is a placeholder for any base language element that is not a syntactic delimiter or a keyword. Thus meta-variables are simply variables that can range over syntactic elements of the analysed language. In addition to meta-variables that have a one-to-one correspondence to individual base language elements, *list meta-variables* can match an arbitrary sequence of elements, e. g. arbitrary many call arguments or statements within a block. Syntactically, meta-variables are denoted by identifiers starting with a question mark, e. g. ?val. List meta-variables start with two question marks, e. g. ??args. Here are two examples:

```
(a)  ?call()        (b) ?called_on.?call(??args)
```

The pattern (a) above specifies method calls without arguments. If evaluated on the program shown in Figure 1 it matches the expressions x.a(), m() and y.b(). For each match of the pattern, the meta-variable ?call is bound to the corresponding identifier (a, m and b). Pattern (b) only matches x.a() and y.b() because it requires the calls to have an explicit receiver. Each match yields a tuple of values (a *substitution*) for the MV tuple (?called_on, ?call, ?args). In our example the substitutions are (x,a,[]) and (y,b,[]), where [] denotes an empty argument list.

B. Element Selection

The predicate is (written in infix notation) enables selection of program elements based on their structure expressed using code patterns:

```
<metavariable> is [[<codepattern>]]
```

```
class A         { void a(){} }
class B exends A{ void b(){} }
class C {
  B m() {
    B x = new B();   // [new B()] <= [x]
    x.a();           // [x] <= A
    return x;        // [x] <= [C.m()]
  }
  void n() {
    B y = m();       // [C.m()] <= [y]
    y.b();           // [y] <= B
  }
}
```

Figure 1. Method invocations, assignments, parameter passing and returns impose constraints on the types [E] of contained expressions E.

The predicate unifies the meta-variable on the left hand side with a program element matched by the code pattern on the right hand side. If the pattern matches multiple elements, each is unified with the corresponding metavariable upon backtracking.

C. Element Context

For many uses, it is not sufficient to consider only a syntactic element itself but also its *static context*. For example, the *declaring type* contains important information about a method or a field declaration. Also the statically resolved binding between a method call and its called method (or a variable access and the declared variable) is necessary for many analyses. This information is available via *context attributes*, which can be attached to meta-variables by double colons. Figure 2 describes the attributes used in this paper.

D. Self-Defined Predicates

The is predicate provides an intuitive way to specify the assumed *structure* of program elements. Context attributes let us concisely express a few *often used relations* between elements. However, for complex analyses, these features need to be complemented by a mechanism for expressing *arbitrary relations* between program elements. Therefore, GenTL lets programmers define their own predicates based on the concepts introduced so far.

Predicates are defined by rules consisting of a left hand side and a right-hand-side separated by ':-'. Multiple rules for the same predicate (that is, with the same left-hand-side) express disjunction. The right-hand-side (the body) of a rule can contain conjunctions, disjunctions and negations. Predicates can be defined recursively, providing Turing-complete expressiveness.

For example, the term 'declaration element' used in [11] denotes the declaration of the static type of methods, parameters, fields and local variables. The predicate decl_element implements this rule, associating each element with its declared type as follows:

?mv::decl	The statically resolved declaration of the element bound to ?mv. Calls reference the called method; variable accesses the declaration of the accessed field, local variable or parameter; type expressions reference a class or interface.
?mv::type	The statically resolved Java type of the expression bound to ?mv.
?mv::encl	The enclosing method or class of a statement or expression bound to ?mv.

Figure 2. Context attributes used in this paper

```
decl_element(?method,?type) :-
 ?method is [[??modif ?type ?name(??par){??stmt}]].
decl_element(?parameter,?type) :-
 ?parameter is [[?type ?name]].
decl_element(?field_or_var,?type) :-
 ?field_or_var is [[?type ?name;]].
decl_element(?elem,?type) :-
 ?elem is [[?type ?name = ?value;]]
```

Each rule describes one possible variant of a declaration element. Each element's structure is specified by a pattern. For instance, the first rule states that the declared type of a method declaration is its return type. The ?method argument of the element predicate called within the rule represents the method declaration. The second argument contains a code pattern describing the structure of method declarations. The pattern contains several meta-variables: ??modif, matching an arbitrary number of modifiers, ?type for the return type, ?name for the method name, ??par for possible parameters and ??stmt for the statements of the method body.

The second clause selects parameter declarations (they are not terminated by a semicolon). The third clause selects field and local variable declarations without an initializer. The fourth one captures initializers. The syntax of code patterns in GenTL generalized the one described in [10].

III. Type Generalization Refactorings and Type Constraints

In this section we introduce by example the challenge of type generalization analysis and the solution approach based on type constraints.

Let us consider the method m in Figure 1. It defines the local variable x to be of type B although only the method a, defined in the more general type A is actually invoked on x. Therefore one might hope to be able to generalize the type of x to A. This would eliminate an unnecessary dependency on a too concrete type. The utility of such dependency reduction becomes obvious if we consider m as a substitute for a whole subsystem that should be decoupled from the subsystem containing the type B.

Unfortunately, the intended generalization is not possible in our example. Method n indirectly enforces the use of B in m: As b is called on y, y has to be of type B. Because the result of m is assigned to y, the return type of m must be B,too.

Finally, x also has to be of type B because it is returned as the result of m.

The approach described in [11] enables us to deduce this relation formally from type constraints implied by method invocations and assignments (including the implicit assignments represented by return statements and parameter passing). The comments in Figure 1, for example, show the constraints for the statements on their left-hand-side. For example the initialization of y with the result of a call to m implies that the return value of m has to be a subtype of y's type ($[C.m()] \leq [y]$). Combination of the inequalities shown in Figure 1 yields the inequality $[x] \leq [C.m()] \leq [y] \leq B$, thus formally proving the necessity of x being of type B.

Summarizing, our example illustrates that

1) the invocation of the method a on x (resp. b on y) implies that the receiver type must be at least A (resp. B);
2) assignments and return types propagate these restrictions to the types of further expressions;
3) based on these constraints we can derive a chain of inequalities proving x must be typed with B, hence cannot be generalized.

In the following section we present the related formal constraints from [11] and our implementation in GenTL[1].

IV. Analysis for Type Generalization

We first implement predicates that capture the type constraints required for our example. Then we show how these predicates can be used to implement the test for non-generalizability.

A. Type Constraints

Method calls. The type of an expression that calls method M must be a subtype of "the most general type containing a declaration of M", denoted $Encl(RootDef(M))$[2]. This is expressed in [11] by the following type constraint:

$$(Call) \quad call\ E.m() \text{ to a virtual method } M \Rightarrow [E] \leq Encl(RootDef(M))$$

We can map the rule (Call) directly to the following rule of the predicate constrained_by_type(?elem,?type), which states that the type of ?elem is at most ?type:

```
constrained_by_type (?elem,?type) :-
  ?call is [[?E.?m(??args)]],
  ?elem = ?E::decl,
  ?M_decl = ?call::decl,
  root_definition(?M_decl, ?rootMethod),
  ?type = ?rootMethod::encl.
```

The first line of the right-hand-side specifies the structure of method calls which the rule is applicable to. The second says that the call constrains the type of the declaration of the message receiver. Unification of two variables is denoted with the infix operator '='. The fourth line determines the root definition of the called method, using the predicate root_definition, which implements the function $RootDef(M)$. The last line says that the type of the message receiver is constrained by the declared type of the root definition. .

Assignment. The type of the right hand side of an assignment must be a subtype of the one of the left hand side:

$$(Assign)\ E1 = E2 \Rightarrow [E2] \leq [E1]$$

This is implemented as a rule for the predicate constrained_by(?e2,?e1). It represents the restriction of the type of the element ?e2 by the declaration of the element ?e1:

```
constrained_by(?E2_decl, ?E1_decl) :-
  ?assign is [[?E1 = ?E2]],
  ?E1_decl = ?E1::decl,
  ?E2_decl = ?E2::decl.
```

Return. The type of a an expression returned by a method must be a subtype of the method's declared type. This is expressed formally as:

$$(Ret)\ return\ E \text{ in method } M \Rightarrow [E] \leq [M]$$

In GentTL, this reads:

```
constrained_by (?E_decl, ?M_decl) :-
  ?return_stmt is [[return ?E;]],
  ?M_decl = ?return_stmt::encl,
  ?E_decl = ?E::decl;
```

B. Test for Generalizeability

The non-generalizability of declarations in a given program P is checked on the basis of the inferred type constraints. The set of non-generalizable elements, $Bad(P,C,T)$, contains all elements of P whose declared type C cannot be replaced with the more general type T. This is the case if the type constraints imply that an element must be typed with a type that is *not* a supertype of T (second line below) or that is a subtype of a non generalizable element (fourth line below):

$$\begin{aligned}(Gen)\quad & Bad(P,C,T) = \\ & \{E \mid E \in All(P,C) \wedge [E] \leq C' \in TC_{fixed}(P) \\ & \quad \wedge \neg T \leq C'\} \cup \\ & \{E \mid E \in All(P,C) \wedge [E] \leq [E'] \in TC_{fixed}(P) \\ & \quad \wedge E' \in Bad(P,C,T)\}\end{aligned}$$

In the above definition, $E \in All(P,C)$ means that E declares an element of type C in program P. $TC_{fixed}(P)$ is the set of type constraints derived for P. The second line corresponds to the test implemented by the predicate constrained_by_type. The fourth line corresponds to the test implemented by constrained_by.

The rule (Gen) is implemented by the predicate not_generalizable(?elem,?generalizedType). It succeeds if

[1]Due to space limitations we omit some details (definition of *root_definition* and handling of multiple subtypes) in the formalism and in our implementation.

[2]We slightly adapted the original notation $Decl(RootDef(M))$ of [11] in order to avoid confusion with the 'decl' context attribute of GenTL.

the declaration of ?elem is not generalizable to the type ?type. Each line on the right hand side of the two implementing rules corresponds to a line of the formal rule (Gen). The two rules express the disjunction in (Gen):

```
not_generalizable(?elem,?generalizedType) :-
  constrained_by_type(?elem,?type),
  !subtype(?generalizedType, ?type).

not_generalizable(?elem,?generalizedType) :-
  constrained_by(?elem,?upper),
  not_generalizable(?upper, ?generalizedType).
```

V. Implementation & Application

The implementation of GenTL is still work in progress. Pattern predicates are successfully implemented in LogicAJ2 [10], a fine-grained aspect-oriented language that is a predecessor of GenTL. GenTL is translated to the logic meta-programming representation supported by the JTransformer system [5], [8]. This mapping is described in [1].

By now, we provide an implementation of type generalization analysis in JTransformer. This analysis has been integrated into our Cultivate plugin for Eclipse [3]. It is run automatically, whenever a source file is saved. Statements that can be generalized are marked and the usual Eclipse 'warning' tooltip indicates the most general types to which they could be generalized. This is illustrated in Figure 3, which shows a slight variation of our example. Here, it is possible to generalize the type of the variable x, the method m and the variable temp to A because b() is not invoked on temp but on the wrapper object y. All lines affected by the possible generalization are highlighted and the ones where generalizations are possible get an additional warning marker.

VI. Conclusion

In this paper we have presented GenTL, an extension of a logic language by a predicate supporting program element selection based on source code patterns containing meta-variables. We have demonstrated that this concept fosters a direct mapping of formal program analysis specifications to their logic based implementation. The formal foundations for GenTL are laid by the theory of logic-based conditional transformation [6], [7], [9]. The implementation of pattern predicates is based on JTransformer [5]. Efficiency and scalability of this system in conjunction with the compilation of logic programs supported by the CTC [2] is demonstrated in [8]. For instance, the identification of all instances of the observer pattern in the Eclipse platform implementation ($\approx$ 11.500 classes) needs less than 8 seconds. Therefore, we think that the design of GenTL opens the door for a desirable mix of high run-time performance and extremely short development time enabled by the high abstraction level supported.

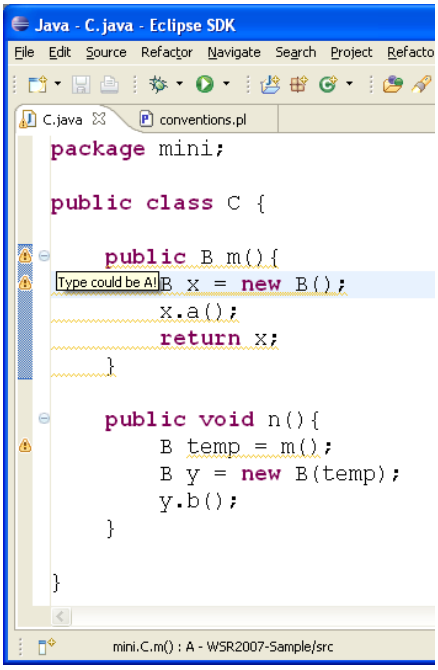

Figure 3. Tool tips indicating possible type generalizations detected by automated logic-based analysis.

References

[1] Malte Appeltauer and Günter Kniesel. Towards concrete syntax patterns for logic-based transformation rules. In *Eighth International Workshop on Rule-Based Programming*, Paris, France, July 2007.

[2] CTC homepage. http://roots.iai.uni-bonn.de/research/ctc/, 2006.

[3] Cultivate homepage. http://roots.iai.uni-bonn.de/research/cultivate/.

[4] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[5] JTransformer homepage http://roots.iai.uni-bonn.de/research/jtransformer/.

[6] Günter Kniesel. A Logic Foundation for Conditional Program Transformations. Technical report IAI-TR-2006-01, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, January 2006.

[7] Günter Kniesel and Uwe Bardey. An analysis of the correctness and completeness of aspect weaving. In *Proceedings of Working Conference on Reverse Engineering 2006 (WCRE 2006)*, pages 324–333. IEEE, October 2006.

[8] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Linking Aspect Technology and Evolution*, March 12 2007.

[9] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming (Special issue on Program Transformation)*, 52(1-3):9–51, August 2004. http://dx.doi.org/10.1016/j.scico.2004.03.002.

[10] Tobias Rho, Günter Kniesel, and Malte Appeltauer. Fine-grained Generic Aspects, Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), AOSD 2006. Mar 2006.

[11] Frank Tip, Adam Kieżun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 13–26, Anaheim, CA, USA, November 6–8, 2003.

[12] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *7th European Conference on Software Maintenance and Reengineering, Proceedings*, pages 91–100. IEEE Computer Society, 2003.

[13] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, 2001.