Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 219 (2008) 113–132
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Towards Concrete Syntax Patterns for Logic-based Transformation Rules

Malte Appeltauer and Günter Kniesel

Dept. of Computer Science III
University of Bonn - Germany
{appeltauer, gk}@cs.uni-bonn.de - roots.iai.uni-bonn.de

Abstract

Logic meta-programming in Prolog is a powerful way to express program analysis and transformation. However, its use can be difficult and error-prone because it requires programmers to know the meta-level representation of the analysed language and to think and express their analyses in terms of this low-level representation. In addition, the backtracking-based evaluation strategy of Prolog may lead to subtle semantic problems when used to express transformations of a logic database. In this paper, we propose an alternative approach, GenTL, a *gen*eric *t*ransformation *l*anguage that combines logic-based *C*onditional *T*ransformations (CTs) and concrete syntax patterns. This combination addresses the above problems while still offering the full expressive power of logic meta-programming. Compared to approaches based on other formalisms, the design of GenTL offers advantages in terms of composability and easy transformation interference analysis.

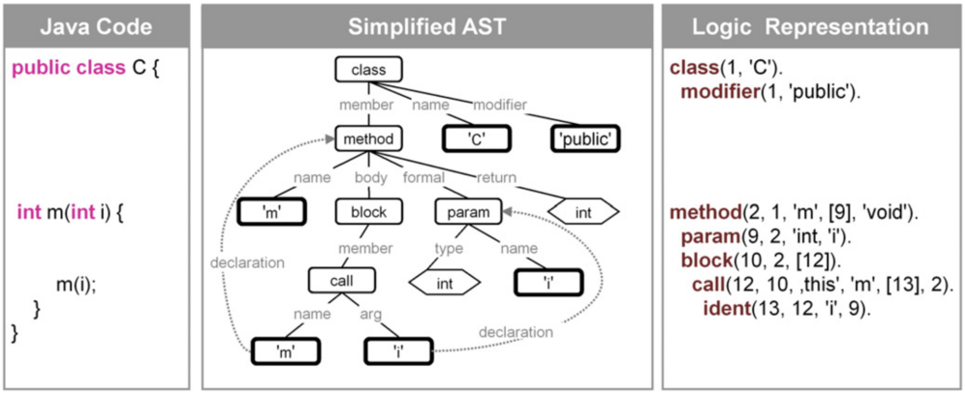
Keywords: Generic transformation rules, concrete syntax patterns, program analysis, GenTL , conditional transformations, logic meta-programming.

1 Introduction

Program transformation, the act of changing one program into another [20], is one of the most fundamental activities of software development. *Automated program transformation* helps software engineers manage the complexity and continuous evolution of modern software projects. In particular, tools that treat transformations as first class entities let programmers quickly define transformations that suit their needs. Depending on their formal foundations these tools can be roughly classified as based on term rewriting [31,4,28,33,1], graph transformation [18], or logic [35,7]. Each category has its particular strengths and weaknesses.

This paper focuses on logic-based approaches, discussing their strengths and showing how they can be further improved by removing two current limitations: the lack of a well-defined semantics and the inability to work with 'concrete syntax', which forces programmers to think in terms of a meta-level representation (Section 2). Section 3 addresses the first problem, presenting *conditional transformations*

doi:10.1016/j.entcs.2008.10.038



```
public class C {

  int m(int i) {

      m(i);
   }
}
```

```
class(1, 'C').
  modifier(1, 'public').

method(2, 1, 'm', [9], 'void').
  param(9, 2, 'int, 'i').
  block(10, 2, [12]).
    call(12, 10, ,this', 'm', [13], 2).
      ident(13, 12, 'i', 9).
```

Figure 1. An object-oriented program, its simplified abstract syntax tree (AST) and the representation of the AST as a set of clauses.

(CTs), a logic-based transformation language with a well-defined formal semantics. Section 4 addresses the second problem, decoupling analyses from the underlying meta-level representation. This is achieved via a new predicate that selects program elements based on source code patterns containing meta-variables ('concrete syntax patterns'). Section 5 presents GenTL a high-level, *gen*eric *t*ransformation *l*anguage that combines the advantages of CTs and concrete syntax patterns. Section 6 compares it to related work. Section 7 discusses future work. Section 8 concludes.

2 Logic Meta-Programming(LMP)

In this section we introduce the idea of a logic-based program representation and the related concept of logic meta-programming (LMP). We discuss the strengths that motivate the use of logic-based approaches and identifying two problems of LMP that will be addressed in the remainder of this paper: too low abstraction level and unclear semantics.

2.1 Logic-Based Program Representation

The commonality of all logic-based transformation approaches is the representation of a program by a set of logic clauses. A possible clausal representation for a small Java program is illustrated in Figure 1. Each clause represents a node of the program's abstract syntax tree (AST). The first argument of the clause is a unique identifier for the node. The other arguments are inlined terminal values or identities of other nodes. Each 'foreign' identity represents a reference to the node with that identity. For instance, the second argument of every non-top-level node is a reference to its enclosing node.

Logic meta-programming is the use of logic programming for analysing and transforming the logic representation of a program [7,35]. The logic programming language is typically a variant of Prolog [3] or some other system implementing top-down SLD resolution [16,15]. Logic meta-programming allows for easy implementation of program analyses as predicates and queries on the program representation.

```
 1 public_self_call(Call,CalledMethod):-
 2   call(Call, _, 'this', _, _, CalledMethod),
 3   modifier(CalledMethod,'public').
 4
 5 introduce_explicit_self(Method,SelfType) :-
 6     method(Method,Class,Name,Params,RetType),
 7     replaceable(Class,SelfType),
 8     create_new_id(Param),
 9     assert(param(Param, Method, SelfType, 'self')),
10     retract(method(Method,Class,Name,Params,RetType)),
11     assert(method(Method,Class,Name,[Param | Params],RetType)).
12
13 redirect_self_calls(Method) :-
14     public_self_call(Call, CalledMethod),
15     enclosing_method(Call, CallingMethod),
16     method(CallingMethod, _, _, [FirstParam | _],_),
17     retract( call(Call, Encl, 'this', Name, Params, CalledMethod) ),
18     assert( call(Call, Encl, FirstParam, Name, Params, CalledMethod) ).
19
20 object_based_inheritance(Method,SelfType) :-
21     add_self_parameter(Method,SelfType),
22     redirect_self_calls(Method).
```

Figure 2. Redirection of self calls via logic meta-programming.

Transformations can be achieved using the meta-programming features of Prolog for asserting and retracting logic clauses.

2.2 *Object-based inheritance via LMP*

Figure 2 illustrates the use of logic meta-programming for transforming the program from Figure 1 into a version that implements the 'passed pointer' design pattern for simulating object-level inheritance in a class-based language. In particular, we focus on the implementation of object-level overriding. The idea is to extend every method by an additional parameter that is henceforth used as the new receiver of previous self-invocations. This lets clients supply a decorator object that overrides behavior of the decorated object [25,10].

The predicate public_self_call implements a simple analysis. It selects all self-invocations of public methods, that is all invocations of public methods whose receiver is this[1].

The predicate introduce_explicit_self implements a transformation that extends methods by an additional parameter of type SelfType. The invocation of replaceable in line 7 checks whether SelfType is a valid replacement for the type of this in the

[1] Prolog variables begin with a capital letter. Underscores denote *distinct* variables whose values are irrelevant.

modified method. Since the type of this is the enclosing class, Class, the replaceable predicate verifies whether SelfType is implemented by Class and provides the same interface as Class. Its implementation is not shown here, for brevity. Line 8 unifies the Param variable with a new, unique identifier. Line 9 creates a clause representing the new parameter. Lines 10 and 11 replace the clause specifying the method signature by a version that includes the new parameter as the first one in the parameter list.

The predicate redirect_self_calls replaces all public self calls by calls on the new 'self' parameter. In line 14 it uses public_self_call to identify places where redirection of the receiver is possible (only invocations of public methods are redirectable). Lines 15 and 16 locate the new parameter as the first parameter of the method that contains the self invocation (the details of redirect_self_calls are omitted for brevity). Lines 17 and 18 replace the invocation on 'this' by an invocation on the parameter.

Finally, the predicate object_based_inheritance invokes the two transformations in the proper order.

2.3 Assessment of LMP

In this section we assess the strengths and weaknesses of LMP with respect to some desirable properties of program transformation systems.

Unified analysis and transformation. Program transformation needs prior program analysis. In all but the simplest cases, analysis is necessary to determine *where* a transformation should take place (e.g. at all public self calls) and *whether* the transformation is legal in that context (e.g. whether the provided SelfType may replace the previous type of this). If existing elements are modified or new ones are added, it is also often necessary to determine the *structure* of the new or modified elements depending on existing elements that are 'far away' in the program [2].

In approaches based on term rewriting, for instance, each of the above steps might require complex traversals of the existing program prior to the transformation, accumulation of relevant information and propagation of the collected information into the transformation. All these functions are smoothly integrated in LMP. Every predicate simultaneously serves all three purposes: determining where, whether and how to transform something. Traversals are just conjunctive conditions. Complex traversals correspond to possibly recursive predicates. Propagation of information from the analysis to the transformation does not require any additional coding of accumulator parameters.

Small Conceptual Gap. For many transformations, the related analyses are much more demanding than the transformations themselves. For instance, Tip, Kiezun and Bäumer [27] show that generalizing refactorings require very complex analyses which in turn need a thorough formal foundation in terms of type constraints [19]. In order to ease the implementation, of such analyses, as well as the comprehension and evolution of the implementation, the implementation language

[2] Consider creation of forwarding methods in a decorator. Their signature and body depends on the public interface of the decorated class.

should be as close as possible to the formal foundation. As logic is the formal basis of many analyses, a logic-based implementation recommends itself as is the most immediate implementation. For instance, Speicher et al. [26] show that a LMP implementation of the work of Tip, Kiezun and Bäumer burns down to just a few lines of code per type constraint, faithfully mirroring the original formalism.

Reuse. It is desirable that program transformations and analyses can be reused as they are to build more complex ones. LMP excels in this domain. First, every predicate with n parameters implicitly defines $n!$ different functions.

For instance, if redirect_self_calls is invoked with a concrete value for its parameter, it will replace just the self calls of that method. If called with a free variable, it will replace all public self calls. Second, propagation of control flow information is automatic and does not require manual encoding. For instance, in object_based_inheritance the second transformation is executed only for those methods for which the first one was successful.

Use of familiar abstractions. Ideally, a program transformation system should let programmers work at the familiar level of abstraction of the transformed language. 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, a logic meta-programming tool for Java [13,9] represents methods by a predicate with 7 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 hides the intention of a transformation behind its meta-level representation. The same critique applies to other representation-centric approaches, e.g. graph transformations.

Well-defined semantics. Last but not least, a program transformation specification should have a well-defined, easy to understand, declarative semantics. Unfortunately, LMP fails to fulfill this essential requirement. The addition or deletion of clauses in a logic program during its execution interferes with the backtracking-based evaluation strategy. This can lead to unexpected results. For instance, the redirect_self_calls in Figure 2 tries to identify self calls in line 14 but also removes self calls in lines 17 and 18. Thus the set of self calls is changed while the determination of self calls is not yet completed (more results to be produced by backtracking).

2.4 Conclusions

Summarizing, LMP offers a smooth integration of analysis and transformation at a high conceptual level and produces easy to reuse implementations. However, it lacks a well-defined semantics and the ability to work with concrete syntax. The remainder of this paper incrementally introduces a logic-based program transformation approach that offers the advantages of logic meta-programming but avoids its problems.

3 Conditional Transformations

Conditional transformations leverage on the power of logic meta-programming by using the same representation of programs a clauses and expressing program analyses as predicates. However, conditional transformation provide an own abstraction for the interplay of analyses and transformations.

A *conditional transformation* (CT) [11,12] is a pair consisting of a precondition C and a transformation T. The precondition can be any logic query that performs no side-effects – in particular no modification of the set of clauses. The transformation is a sequence of the following basic actions:

- skip does nothing
- add adds a new clause
- delete deletes a clause
- replace replaces a clause with another one

Application of a conditional transformation to a program P consists in (1) determining the set of all substitution for the variables in C that make C true in P, (2) applying each of the computed substitutions to T, and (3) executing all the resulting transformations on P. This separation of precondition evaluation from transformation execution avoids the semantic problems of logic meta-programming.

Figure 3 presents the CT-based implementation of the example from 2. The 'only' distinction to the LMP version is the clear separation between conditions and transformations and the resulting well-defined semantics of CT application.

As shown in the last three lines of Figure 3 complex transformations can be expressed as CT sequences using two composition operators, AND-Sequence and OR-Sequence. In addition, sequences can be executed via the LOOP operation until a fixpoint is reached, that is until the preconditions of all CTs in the sequence fail. For a detailed discussion of CTs, CT sequences and their precise formal definition see [11].

In an OR-Sequence, every transformation is executed for all the substitutions that make its precondition and all the preceding preconditions true. Thus results of previous CTs influence latter ones but not vice versa.

In contrast, AND-sequences express a mutual dependency of all CTs in the sequence: if there is a substitution that makes *all* the preconditions in the *entire* sequence true, then all the associated transformations are executed in the specified order. Otherwise, none of them is executed or those that have already been executed are undone. Thus AND-sequences behave like fine-grained transactions. In Figure 3 the use of an AND-sequence (line 27 and 28) ensures that addition of parameters is undone for methods for which redirection of self calls fails. For all the other methods the transformation is executed normally.

AND-sequences are a unique feature of CTs. No program transformation system we know of provides a similar functionality. Another distinguishing feature is that CTs enable easy and efficient interference analysis of independently developed but jointly deployed transformations. Mens et al. show that the only other system that

```
ct( add_self_parameter(Method,SelfType),
     forall( (
        method(Method,Class,Name,Params,RetType),
        replaceable(Class,SelfType) ) ),
    do( (
        create_new_id(Param),
        add(param(Param, Method, SelfType, 'self')),
        replace( method(Method,Class,Name,Params,RetType),
                 method(Method,Class,Name,[Param | Params],RetType) ) ) )
).

ct( redirect_self_calls(Method) ,
    forall( (
       public_self_call(Call, CalledMethod),
       enclosing_method(Call, CallingMethod),
       method(CallingMethod, _, _, [FirstParam | _],_) ) ),
    do( (
       replace( call(Call, Encl, 'this', Name, Params, CalledMethod),
                call(Call, Encl, FirstParam, Name, Params, CalledMethod) ) ) )
).

ctseq( object_based_inheritance(Method,SelfType) :
       add_self_parameter(Method,SelfType) AND
       redirect_self_calls_in(Method) .
```

Figure 3. Redirection of self calls using CTs and CT sequences.

achieves a similar interference analysis is four orders of magnitude slower [17].

CTs and CT sequences can express influential concepts of modern software engineering. A refactoring, for instance, is just a behaviour preserving conditional transformation [14]. Similarly, aspects can be expressed as sequences of CTs. Windeln [32] shows that the generic aspect language LogicAJ can be compiled to CTs. Conditional transformations are supported by *JTransformer* and the *CTC* (Conditional Transformation Core) system. JTransformer [9,13] is a logic-based source-to-source transformation system for Java. It maps programs to a logic fact representation and supports conventional logic meta-programming in Prolog (see Section 2) as well as conditional transformations and a limited form of CT sequences. The CTC [6] is a complete, language-independent implementation of CTs, CT sequences and the related interference analyses [12].

Their expressive power, well-defined semantics, and unique features recommend CTs as a basis for complex program transformations. However, working at AST level, they are better suited as a formal foundation than as a language for practical use. The remainder of this paper shows how the level of abstraction can be raised by the use of 'concrete syntax'.

4 Program Analysis with pattern predicates

This section introduces our combination of logic-based program analysis with concrete syntax patterns. The relevant concepts are illustrated using Java as source and target language. Section 7 discusses a language-independent generalization of our transformation system.

4.1 Concrete Syntax Patterns and Meta-Variables

A *concrete syntax pattern* (CSP) is a snippet of concrete base language code, e.g., Java, that may contain meta-variables. A *meta-variable* (MV) is a placeholder for any legal expression of the base language, i.e, all generating non terminal symbols of the language's BNF representation. Thus, meta-variables are simply variables that can range over syntactic elements of the analyzed language. Normal meta-variables represent a single base language element. In addition, *list meta-variables* can match an arbitrary sequence of elements of the same sort, 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.

Consider the example shown in Figure 4 and assume we want to select all method invocations. This can be achieved via the following concrete syntax pattern pattern capturing the structure of method calls in Java:

```
?expression.?call(??args)
```

If evaluated on the program shown in Figure 4 it matches the calls in Lines 3 to 5. For each match of the pattern, the meta-variable ?call is bound to the corresponding method name (a and y) whereas ?expression is bound to the expression denoting the object on which the method is invoked: A, this, y() and this. The latter is the implicit receiver of the first call in Line 5. For each match, the list meta-variable ??args is bound to the call arguments. In our example only the call in Line 3 contains arguments. In all other cases ??args is bound to an empty argument list. Each match of the CSP yields a tuple of values (a substitution) for the MV tuple (?expression, ?call, ??args). In our example the substitutions are (A,a,[42,43]), (this,y,[]), (this,y,[]) and (this.y(),y,[]), where [] denotes the empty list.

```
1  class C{
2    void x(){
3      A.a(42,43);
4      this.y();
5      y().y();
6    }
7    C y(){
8      return this;
9    }
10 }
```

Figure 4: A simple Java class

4.1.1 Structural pattern matching

Concrete syntax patterns are matched at the AST level. Except for the use of meta-variables, a pattern must correspond to a valid AST of the base language. The pattern's AST is matched to the base program's AST. Meta-variables of the CSP can match entire subtrees within the base program.

Figure 5 shows the matching of our example pattern at the AST level. First,

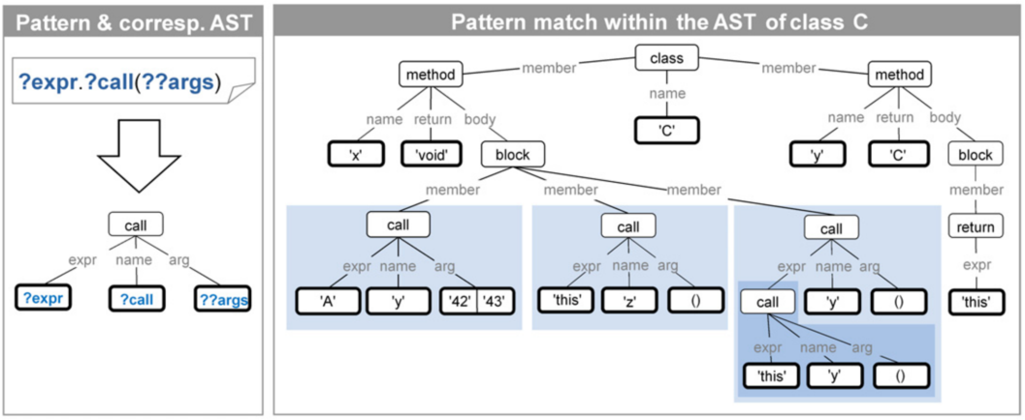


Figure 5. Pattern matching at AST level

the pattern is transformed into its corresponding AST. This pattern matches four subtrees within the AST of C. The blue boxes indicate the matches. Note that a pattern can match recursively within an subtree. This is the case for the expression y().y() in Line 4 of Figure 4. The two nested blue boxes (the rightmost boxes) in Figure 5 represent the recursive match.

Since CSPs match at the AST level, matching is not restricted to lexical structures. For instance, the pattern

```
if(?expr){ ??statements }
```

matches the statement

```
if(a < b) a=b;
```

although its condition is not enclosed in curly braces.

Patterns must be complete and valid expressions of the base language. However, a pattern does not need to specify all elements of the matched element. Consider, for instance, the pattern

```
class ?classname{??class_members}
```

It contains neither a modifier, nor a declaration of interface implementation, nor a superclass declaration. Still, it matches both of the following:

```
public class C implements D {...}
abstract_public class C extends E {...}
```

4.2 Pattern predicate

In order to use CSPs in logic programms, we introduce a special *pattern predicate*, is. It is often necessary to be able to refer to the entire program element matched by a CSP. For this purpose, the left-hand-side of is denotes the program element matched by the CSP on the right-hand-side.

meta-variable is [[*concrete_syntax_pattern*]]

Technically, the predicate unifies its first argument with the unique identifier of a program element matched by the syntax pattern in the second argument. If the pattern matches different elements, each one's identity is unified with the meta-variable upon backtracking. The pattern predicate combines the expressiveness of logic programming with intuitive pattern description.

It is often 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 analyzes. This information is available via *context attributes*, which can be attached to meta-variables by two double colons. In this paper, the following attributes are used:

?mv::decl The statically resolved corresponding declaration of ?mv. Calls reference their method, variable accesses a field, and local variable or parameter declarations and type expressions reference their class.

?mv::type The statically resolved Java type of an expression bound to ?mv.

?mv::encl The enclosing method or class of a statement or expression.

?mv::parent The parent element of ?mv.

Note that meta-variable attributes can be cascaded. For example, ?mv::encl::type refers to the type of the enclosing method or class of the element bound to ?mv.

4.3 Self-Defined Predicates

The pattern 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 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 rules can contain conjunction, disjunction and negation. Predicates can be defined recursively, providing Turing-complete expressiveness.

4.4 Example

We illustrate the use of pattern predicates with a simple example. Assume we want to select all method calls within the program of Figure 4 that invoke a method of their enclosing class C. Figure 6 shows a predicate to express this analysis. The use of the pattern predicate in Lines 2-3 binds method calls to the variable ?call. The second one (Lines 6-10) matches public methods. Line 5 uses the unification operator '=' to expresses that the method invoked by ?call must be the public one matched by the second concrete syntax pattern.

If evaluated on the program from Figure 4 public_self_calls binds its argument

```
public_self_calls(?call, ?method):—
  ?call is [[
    this.?name(??args)
  ]],
  ?method = ?call::decl
  ?method is [[
     public ?type ?name(??params){
        ??stmts
     }
  ]].
```

Figure 6. Program analysis for self calls

tuple (?expr, ?name) to the result tuples (this, y), (y,y) and (this, y).

5 GenTL: Conditional Program Transformations with Concrete Syntax

In this section, we give an overview about GenTL (Generic Transformation Language). GenTL combines the benefits of concrete syntax based program analysis and conditional transformations. The main features of GenTL are:

High-level CTs GenTL supports conditional transformations and conditional transformation sequences as first-class language constructs.

Logic based analysis Self-defined, side-effect free predicates can be used to express arbitrary program analyzes for CT preconditions. In these predicates, concrete syntax patterns can be used to select elements of base programs.

Generic transformations Concrete syntax patterns containing meta-variables can be used uniformly in predicates and transformations. The use of the same meta-variables links preconditions and transformations in ct declarations.

This design provides benefits in terms of reduced dependencies on transformed programs and increased reusability of transformation specifications. Because the meta-variable values are determined by evaluation of predicates and preconditions can share meta-variables with transformations, it is possible to define analyses and transformations without referring to concrete instances, e.g., identifier or types of program elements. This leads to very generic and reusable transformation specifications. In addition, GenTL provides reusability of the elements used in analyses and transformations. Predicates encapsulate program analyses that can be reused in different preconditions. The generate declarataion encapsulates concrete syntax based templates for the code to be generated by transformations.

5.1 Syntax

GenTL adopts the syntax for meta-variables and concrete syntax patterns introduced in Section 4.

MetaVariable := ?*Name* | ??*Name*
CSP := '[[' ... base language code with meta-variables... ']]'

5.1.1 Predicates

GenTL supports self-defined logic predicates in the usual Prolog notation. Own predicates can be defined based on a standard set of predefined predicates common to logic programming languages. However, GenTL explicitly excludes predicates that change the logic database. Only the most important predefined predicates are described in the following.

For encapsulating concrete syntax patterns GenTL supports the predicate is, described in detail in Section 4. Unification of two variables is denoted with the infix operator '='.

PatternPredicate := *MetaVariable* is *CSP*
Unification := *MetaVariable* = *MetaVariable*

If the arguments are different constants of variables bound to different constants, unification fails. Otherwise the arguments are unified, that is, they are bound to the same value. Note that, unlike an assignment, unification is symmetric. For instance, the unification of a constant on the left-hand-side and a variable on the right-hand-side, e.g. const = ?var succeeds if the variable has the same value as the constant or did not have a value before.

The predicate member(*element, list*) checks if an element is a member of a given list. Elements can be removed from lists with delete(*old_list, element, new_list*).

5.1.2 Transformations

A conditional transformation is a first-class language element in GenTL. Its syntax is described below in EBNF notation. Non-terminal symbols are indicated by *italics*. Terminal symbols are in sans serif font. Note that ':=', '—', '[', ']', and '*' are the usual symbols of the EBNF notation whereas '[[' and ']]' are terminal symbols of GenTL:

```
CT := ct Name ( Arguments ) : Condition -> Transformation .
Arguments := [ MetaVariable [ , MetaVariable ]* ]
Condition := PredicateInvocation
                     | ! Condition                    // Negation
                     | Condition , Condition          // Conjunction
```

```
                  | Condition ; Condition          // Disjunction
Transformation := Action [, Action]*
```

CTs are well-formed if and only if their transformation part only contains meta-variables that are also part of the condition. A condition is any logic expression that can be built by conjunction, disjunction and negation from the self-defined and predefined predicates. A transformation is a sequence of the following basic actions:

```
Action := skip
        | add Template before MetaVariable
        | add Template after MetaVariable
        | delete  MetaVariable
        | replace MetaVariable with Template
Template := [[ ... base language code with meta-variables... ]]
```

Compared to the basic CT concept, the syntax of transformations supports a higher level of abstraction. First, it enables use of concrete syntax patterns for specifying code to be generated. Second, it frees programmers from having to manage the creation of new unique element identities. Third, it allows for specifying relative locations of added elements with respect to existing ones.

The first two feature free GenTL programmers from having to know the internal representation of a program. As a consequence, they cannot insert elements at a specific location by setting the respective arcs in the (unknown) AST. This explains the necessity of a syntax for specifying relative locations independent of the internal representation.

In order to support reuse, source code to be generated by transformations can be encapsulated into named and parameterized templates using generate declarations. This corresponds to the following refinement of the definition of templates:

```
Template := CSP | Generate
Generate := generate Name ( Arguments ) CSP
```

The signature (name and arguments) defined by a generate declaration can be used like a concrete syntax pattern. It acts like a macro, propagating its arguments into the patten that it encapsulates.

Transformations can be composed using AND- and OR-Sequences:

```
CTSEQ := ctseq Name ( Arguments ): SeqBody .
SeqBody := CT_Call
         | CT_Call OR SeqBody
         | CT_Call AND SeqBody
CT_Call := Name ( Arguments )
```

5.2 *Example: Object-based inheritance*

Section 2.2 introduced transformations needed to achieve object-based inheritance and demonstrated in Figure 2 their implementation via logic meta-programming. Figure 3 demonstrated their CT-based implementation. In this section we present the implementation of object-based inheritanace in GenTL. Figure 7 shows the necessary two conditional transformations and the CT sequence combining them. The GenTL definition of the public_self_call predicate has already been presented in Figure 6 on page 11. The implementation of the predicate replaceable follows the same schema and has been omitted for brevity.

The CT add_self_parameter introduces a new parameter named self of type ?selfType to all public methods for which the replaceable test in line 4 is true (cf Section 2.2)[3]. The add transformation expresses that the new parameter is inserted as the first one in the list of parameters of the modified method.

The CT redir_self_calls_in redirects all this-calls to the new self parameter. The precondition matches all this-calls, making their receiver, ?recv, which is bound to this, available for replacement. The action replaces ?expr with self, the identifier of the new parameter.

Finally, both transformations are combined to an *AND* sequence ensuring that the transformation is executed only on those methods for which both CTs succeed.

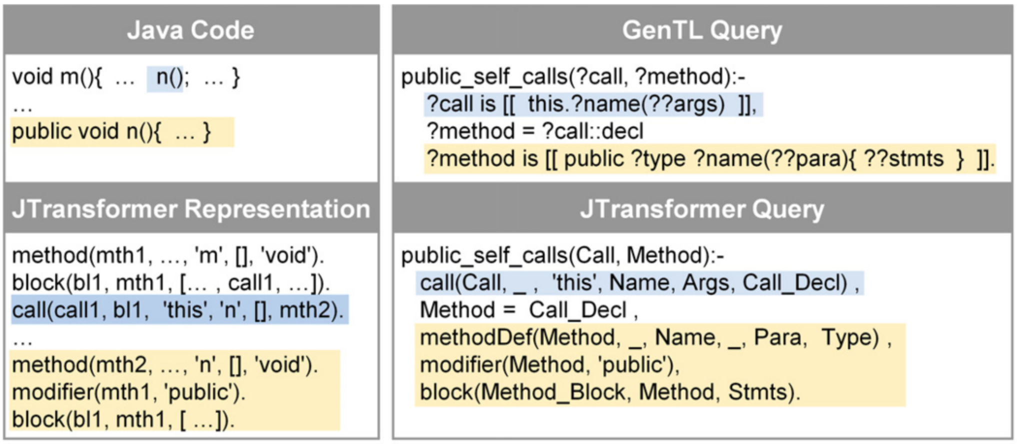
5.3 *Implementation Scheme*

The implementation scheme for GenTL is build upon transforming concrete syntax patterns to an equivalent logic query, using JTransformer as a back-end. Figure 8

[3] We assume that the invocation of the transformation provides a binding for ?selfType.

```
 1 ct add_self_parameter(?method, ?selfType):
 2   ?method is [[  public ?type ?name(??params){ ??statements }  ]],
 3   ?type = ?method::encl::type,
 4   replaceable(?type,?selfType),
 5  ->
 6   add [[?selfType self]] before ??params .
 7
 8 ct redir_self_calls_in(?calledMethod):
 9   public_self_call(?call, ?calledMethod),
10   ?call is [[?recv.?name(??params)]]
11  ->
12   replace ?recv with [[self]] .
13
14 ctseq use_self_param(?method, ?selfType):
15   add_self_parameter(?method, ?selfType) and
16   redirect_self_calls_in(?method).
```

Figure 7. Redirection of self calls in GenTL. See Figure 6 for the definition of the predicate public_self_call.



```
void m(){ ...  n();  ... }
...
public void n(){  ... }
```

```
public_self_calls(?call, ?method):-
   ?call is [[  this.?name(??args)  ]],
   ?method = ?call::decl
   ?method is [[ public ?type ?name(??para){ ??stmts  } ]].
```

```
method(mth1, ..., 'm', [], 'void').
block(bl1, mth1, [... , call1, ...]).
call(call1, bl1,  'this', 'n', [], mth2).
...
method(mth2, ..., 'n', [], 'void').
modifier(mth1, 'public').
block(bl1, mth1, [ ...]).
```

```
public_self_calls(Call, Method):-
  call(Call, _ ,  'this', Name, Args, Call_Decl) ,
  Method =  Call_Decl ,
  methodDef(Method, _, Name, _, Para,  Type) ,
  modifier(Method, 'public'),
  block(Method_Block, Method, Stmts).
```

Figure 8. *left:* Mapping of Java code to its JTransformer representation. *right*: A GenTL pattern predicate and the JTransformer representation of its concrete syntax pattern. Meta-variables are denoted as unbound variables in the internal view.

illustrates side-by-side the translation of Java to logic facts and the translation of GenTL concrete syntax patterns to an equivalent logic query.

The left part shows a Java code snippet and its logic fact representation. Use of a color in the lower part of the figure indicates the internal representation of the program element marked with the same color in the upper part.

The right-hand-side shows a pattern predicate and its implementation as a JTransformer query. Here, use of a color in the lower part indicates the implementation of the GenTL expression with the same color from the upper part. Use of a color on the right-hand-side indicates a GenTL expression (or its implementation) that matches the Java element (or logic facts) marked with the same color on the left-hand-side.

Note that GenTL meta-variables are directly mapped to unbound logic variables in the implementing JTransformer query. Note also that the illustrated translation of Java to logic facts also defines the translation of concrete syntax patterns used in transformations. At the time when a transformation is executed, all the meta-variables in its patterns are guaranteed to be bound to concrete values by the prior evaluation of the condition. Thus, the patterns have become plain base language code.

6 Related Work

The **TyRuBa** language [7] introduces logic meta-programming for Java. It supports Prolog-like predicates that express analyzes on Java programs and others that generate Java code. Code generation for method bodies (blocks) is based on 'quoted expressions'. These are strings that can contain meta-variables for types and identifiers. The syntactic type of meta-variables must be explicitly declared. Quoted expressions can be regarded as a strongly restricted form of concrete syntax patterns. They are only used for generating Java code but not in the query language.

SOUL, the Smalltalk Open Unification Language [35], enables meta-programming with logic queries and program transformations for Smalltalk, in Smalltalk. It provides a sophisticated, dynamic integration of the object-oriented base language and the logic-based meta-language, which goes beyond the capabilities of GenTL. However, its ability to express concrete syntax patterns is restricted to quoted strings, as in TyRuBa. Recently, Roover et al. [24] presented an extension of SOUL for matching *execution flows* with concrete syntax patterns.

For instance the pattern `void a(){b();}` matches a dynamic call of `b()` that occurs within the control flow of `a()`. However, they are using their system only for program analysis, not transformation. GenTL supports both and deliberately limits itself the more basic structural matching, giving programmers the ability to build their own abstractions in terms of data-flow, control-flow, etc. by implementing suitable analysis predicates.

All systems mentioned so far support different variants of logic meta-programming based on SLD resolution. As such they all suffer from the semantic problems described in Section 2.3. An exception from this rule is **JTransformer** [9,13], a logic-based source-to-source transformation language for Java which supports logic meta-programming *and* conditional transformations. Due to its ability to execute CTs, JTransformer has been used as a basis for implementing the generic aspect language **LogicAJ** [21,32]. LogicAJ supports uniform genericity [8] by the use of meta-variables in predicates *and* code generation templates based on concrete syntax. However, use of concrete syntax is not possible in LogicAJ's predicates.

Its successor, **LogicAJ2** [23] supports *fine-grained* aspect genericity. Its join point [4] model covers all syntactic elements of a base language. Easy join point selection is enabled by three build-in predicates that specify selection criteria for declarations, statements and expressions based on concrete syntax patterns. In GenTL these predicates are generalized and unified into a single pattern predicate, `is`. This simplifies the language and frees programmers of thinking in terms of syntactic categories.

The **ASF+SDF meta-environment** [28,29] is a system for parser generation and program transformation. Next to a source language specification with ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism) one can define equations simplifying expressions written in the base language. This is a powerful technique since it allows a complete evaluation of the program. ASF+SDF does not support concrete syntax patterns.

Borba et al. [2] introduce **JaTS**, the Java Transformation System, which allows the definition of pattern based rewriting rules. Each rewriting rule is a pair of concrete syntax patterns describing the program state before and after a transformation. Like GenTL, both parts can be linked by the use of common meta-variables

[4] In aspect-oriented programming, join points denote well defined points in the structure or execution of a program. For the context of this paper, join points can be regarded as program elements.

that substitute syntactic elements [5] . However, JaTS does neither support logic expressions nor traversal strategies, which would allow complex queries or composition of transformations.

Stratego [31,30] is a term rewriting system offering various generic term traversal *strategies*. Strategic rules allow the user explicitly to control rule application and term traversal. In order to parse and transform programs of a language L the system requires a SDF representation of L. Thus it inherits some of the abilities of SDF and related distinctions to GenTL. In addition, Stratego supports concrete syntax patterns within matching parts and transformations, much like GenTL.

MetaBorg [1] extends Stratego with domain specific language embedding, that is, the ability to manage a host language and additionally an embedded language. Definition of assimilation rules with concrete syntax enables a mapping of program elements of the embedded language to host language elements.

The **TXL** [5,4] system was originally built for rapid prototyping. Features and behavior of TXL transformations are very similar to Stratego. Like Stratego, TXL transformations can be specified with concrete source code for pattern matching and transformation.

HATS, the High Assurance Transformation System [33,34], is a strategic term rewriting system for manipulating parse trees. It offers various combination operators and generic traversal strategies. Its specific strengths is that it enables program transformations produce programs with *provably correct syntax*. This is something that GenTL does not provide yet. HATS does not support concrete syntax patterns, e.g., on the left-hand-side of rewrite rules.

All systems listed above, except JaTS, are language independent, that is, they can be applied to arbitrary base languages specified by a grammar in an EBNF style notation. Their parser generation ability is lacking currently in GenTL and is one of the main extensions planned for future work.

Another main distinction between ASF+SDF, JaTS, Stratego, MetaBorg, TXL, HATS, one one hand, and GenTL, on the other, stems from the differences between a term rewriting system and the logic-based, universally quantified evaluation of CTs. CTs and the CT sequence operators provide easy to implement, reusable abstractions for determining where, whether and how to transform *all* elements that satisfy some arbitrarily complex condition. Expressing the same functionality in a term rewriting system requires a lot of additional coding, e.g. complex traversals of the existing program for verifying non-local conditions, further traversals for accumulating information relevant for the transformation, propagation of the collected information into the transformation nd propagation of information from one transformation to the other. See also the first discussion in Section 2.3.

[5] [2] only describes matching of interface-level program elements. However, according to personal communication with the authors, meta-variables can also match finer-grained elements.

7 Future Work

In this paper we present GenTL as a transformation language for Java. Nevertheless, its core concepts are language-independent. Therefore, GenTL will be extended with a generic compiler for arbitrary language specifications, following the example of the transformation systems discussed above.

The components of generic compilers are generally introduced in [30].

Static analysis of concrete syntax patterns is difficult because of lacking information about the values of meta-variables. In order to prevent that, for instance, a transformation substitutes an expression for a statement, one needs to know at least the*syntactically type* of meta-variables.

Like its predecessor, LogicAJ2, GenTL provides a way to explicitly declare syntactic types for meta-variables. We have omitted discussing this feature because we regard it as a step back to a language model that forces programmers to know and think in terms of the entire set of syntactic categories of the base-language. Rho and Kniesel [22] propose a type inference mechanism for LogicAJ that we intend to adapt and generalize for GenTL. Another source of inspiration could be HATS [33,34].

8 Conclusions

Defining complex analysis and transformation rules using logic meta-programming can be elaborate, error-prone and suffers from semantic problems. The *conditional transformation* formalism supports logic program transformation with a structural breakup of analysis and transformation. Additionally, CTs provide a composition mechanism for transformations. However, CTs are still declared at the AST level. We claim this level provides not the appropriate abstraction for transformations.

In this paper, we have introduced GenTL, a generic transformation language. GenTL combines concrete syntax patterns with predicates. Transformations are defined as conditional transformations. As a result, GenTL allows the definition of program analysis and transformation rules in a easy and reusable way.

References

[1] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[2] Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. JaTS: A Java transformation system. In *XV Brazilian Symposium on Software Engineering*, pages 374–379, October 2001.

[3] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog (3rd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[4] James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[5] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Software Engineering by Source Transformation-Experience with TXL. In *SCAM*, pages 170–180, 2001.

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

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

[8] Tobias Rho Günter Kniesel. A definition, overview and taxonomy of generic aspect languages. *L'Objet, Special issue ' Développement de logiciels par aspects'*, 12(2-3):9–39, September 2006.

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

[10] Günter Kniesel. *Darwin – Dynamic Object-Based Inheritance with Subtyping.* PhD thesis, CS Dept. III, University of Bonn, Germany, 2000.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Robert Kowalski. A proof procedure using connection graphs. *J. ACM*, 22(4):572–595, 1975.

[16] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.

[17] Tom Mens, Günter Kniesel, and Olga Runge. Transformation dependency analysis - a comparison of two approaches. *L'Objet, Special issue 'Langages et Modèls à Objets 06'*, 12(HS):167–182, January 2006.

[18] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 2006.

[19] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley and sons, 1994.

[20] http://www.program-transformation.org.

[21] Tobias Rho and Günter Kniesel. Uniform Genericity for Aspect Languages. Technical report IAI-TR-2004-4, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, December 2004.

[22] Tobias Rho and Günter Kniesel. Syntactic Type Safety for Context-Dependent Generative Programming. submitted to Generative Programming and Component Engineering (GPCE'07), 2007.

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

[24] Coen De Roover, Johan Brichau, Carlos Noguera, Theo D'Hondt, and Laurence Duchien. Behavioral Similarity Matching using Concrete Source Code Templates in Logic Queries. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM07 - co-located with POPL07)*, 2007.

[25] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for self. *Lecture Notes in Computer Science*, 952:pp. 303ff., 1995.

[26] Daniel Speicher, Malte Appeltauer, and Günter Kniesel. Code Analysis for Refactoring by Source Code Patterns and Logical Queries. submitted to 1st ECOOP Workshop on Refactoring Tools, 2007.

[27] 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.

[28] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[29] Mark G. J van den Brand and P. Klint. ASF+SDF Meta-Environment User Manual. Technical report, Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, January 2005.

[30] Eelco Visser. Meta-Programming with Concrete Object Syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[31] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[32] Tobias Windeln. LogicAJ - Eine Erweiterung von AspectJ um logische Meta-Programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany, August 2003.

[33] Victor Winter and Jason Beranek. Program Transformation Using HATS 1.84. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 378–396. Springer-Verlag, 2006.

[34] Victor L. Winter. An Overview of HATS: A Language Independent High Assurance Transformation System. In *Proceedings of IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET).*, pages 222 – 229, March 1999.

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