Program Transformation Mechanics

A Classification of Mechanisms for Program Transformation
with a Survey of Existing Transformation Systems

Jonne van Wijngaarden
Eelco Visser

UU-CS-2003-048
Institute of Information and Computing Sciences
Utrecht University

May 2003

Address:
Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80089
3508 TB Utrecht

Eelco Visser <visser@acm.org>
http://www.cs.uu.nl/people/visser

Program Transformation Mechanics

A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems

Jonne van Wijngaarden and Eelco Visser

Institute of Information and Computing Sciences, Utrecht University, P.O.Box 80089, 3508 TB Utrecht, The Netherlands.
visser@acm.org, http://www.cs.uu.nl/~visser

May 2003

Abstract

Transformation techniques are spreading from application in compilers to general use in generative programming and document processing. Since transformation requires operations such as pattern matching, generic structure traversal, and querying, which are not normally provided by general-purpose programming languages, many tools have been developed to provide higher-level support for the implementation of transformations. These tools come in many flavors each with their own merits and based on different paradigms, which makes comparison difficult.

In this paper, we consider transformation from the point of view of *mechanics* and develop a classification of transformation mechanisms that provides a reference for comparing tools developed for different applications, using different implementations, and in different programming paradigms. To do so we distinguish three fundamental aspects of transformation mechanisms: *scope*, *direction*, and *stages*. We apply this classification in a discussion of design patterns for transformation, characterization of several typical transformations, and a systematic comparison of eleven representative transformation tools.

1 Introduction

Program transformation, i.e., the automatic manipulation of source programs, emerged in the context of compilation, as a set of techniques for the implementation of components such as optimizers [20]. While compilers are rather specialized tools developed by few, transformation systems are becoming widespread. In the paradigm of generative programming [7], the generation of programs from specifications forms a key part of the software engineering process. In refactoring [10], transformations are used to restructure a program in order to improve its design. Other applications of program transformation include migration and reverse engineering. The common goal of these transformations is to increase programmer productivity by automating programming tasks.

With the advent of XML, transformation techniques are spreading beyond the area of programming language processing, making transformation and generation a necessary operation in any scenario where structured data play a role.

Techniques from program transformation are applicable in document processing. In turn, applications such as Active Server Pages (ASP) [19] for the generation of web-pages in dynamic HTML has inspired the creation of program generators such as Jostraca [21], where code templates specified in the concrete syntax of the object language are instantiated with application data.

Since transformations are operations on structured data, they can be implemented in any programming language with data structure support. However, transformation requires operations such as pattern matching, generic structure traversal, and querying, which are not normally provided by programming languages. Directly implementing these operations leads to low level code, which is hard to develop and maintain. Consequently, many tools have been developed to provide higher level support for the implementation of transformations. Such transformation tools come in many flavors, which can be accounted for by differences in application area, implementation decisions, and programming paradigm. A good understanding of these differences is key for designing transformations and for comparing transformation tools to select an appropriate tool for a project.

The *application area* is the prime determinant for choices in the expressiveness and usability of a tool. Many tools are specialized for one or a few application areas and provide an ad hoc solution to a specific problem, while others provide a more widely applicable generic solution. Also different application areas may pose conflicting requirements on transformation tools. For example, an application generator, which generates source code from some form of specification, is required to produce human readable source code. If the generation can be performed in a single stage, there may be no harm in representing the target program as a list of strings. On the other hand, an optimizing component of a compiler should transform a program in some intermediate tree structure produced by a front-end and is not required to produce readable code. References to differences caused by application area can also be found in [7].

Implementation decisions are another source of differences between tools. Many seemingly irrelevant implementational differences like the choice for a certain parser or code representation can have a consirable impact on the expressiveness and usability of a tool. For example, the ATerm API and exchange format [24] unifies the internal and external representation of abstract syntax trees, making it very easy to divide a transformation system into reusable components. On the other hand, the datatype facilities of general purpose programming languages do not support cheap marshalling and unmarshalling, which makes componentization much less natural.

The *programming paradigm* underlying the specification language used by a transformation tool also has considerable impact on the way transformations are developed. For example, pattern matching is natively supported in term rewriting and functional programming with algebraic data types. On the other hand, recognizing complex program patterns in an object-oriented program is much harder.

While these differences explain historical development and popularity of a tool for certain applications, they obscure the fundamental choices that underly the design of a transformation tool, lacking a terminology for describing the properties of transformations. For example, a transformation implemented using an attribute grammar evidently has a different style from a similar transformation using strategic programming, but there are no systematic studies into

these differences.

In this paper, we consider transformation from the point of view of *mechanics* and develop a classification of transformation mechanisms that provides a reference for comparing tools developed for different applications, using different implementations, and in different programming paradigms. To do so we distinguish three fundamental aspects of transformation mechanisms: *scope*, *direction*, and *stages*. The *scope* of a transformation (Section 2) concerns the areas of a program that are affected by a transformation and from which information is used. The *direction* of a transformation (Section 3) determines whether the shape of the source or the target program drives the transformation process. The *staging* of a transformation (Section 4) determines the phases or iterations required or used.

The idea of distinguishing transformation mechanisms is not completely new. For example, the ASF+SDF rewrite system distinguishes several styles of transformations through the mechanism of traversal functions [25], i.e., transformers, accumulators, and accumulating transformers. While these styles can also be recognized in other program transformation tools, this differentiation is too limited as a basis for comparing transformations or transformation tools. The survey of rewriting strategies in program transformation systems [28] focuses on the control over the application of transformation rules specified using rewrite rules. Here the focus is on the mechanisms behind the specification and implementation of individual transformation steps, rather than the control mechanisms.

The contribution of this paper is twofold. First we develop terminology for discussing transformations and transformation tools separately from implementation details. For developers of program transformations, this will be useful in assesing tools and for designing transformations at an abstract level. For tool developers this terminology will be useful in documenting the types of transformation supported by their tool. The current diversity of terminology used in the literature makes it difficult to determine the capabilities of tools. The transformation mechanisms introduced in this paper are a first attempt at establishing such an idiom.

Secondly, we apply this terminology in the analysis of typical transformations and transformation tools. In Section 5 we consider several transformation *design patterns* that arise as combinations of the various mechanisms. Furthermore, we consider several concrete transformations and describe their mechanics. In Section 6 we examine the coverage of the transformation mechanisms by a number of well known transformation paradigms and tools to evaluate the suitability of different paradigms and tools.

2 Scope

Scope literally means *the area covered by a given activity or subject.* In the context of program transformation we define scope as the area of an object program covered by a single *transformation step.* A transformation step can be defined as the single application of a transformation rule. A complete transformation consists of one or more transformation steps. This is typically implemented as some kind of traversal that applies transformation rules at multiple locations. As an example consider a transformation that performs method inlining. A single transformation step performs the inlining of a method body at one call

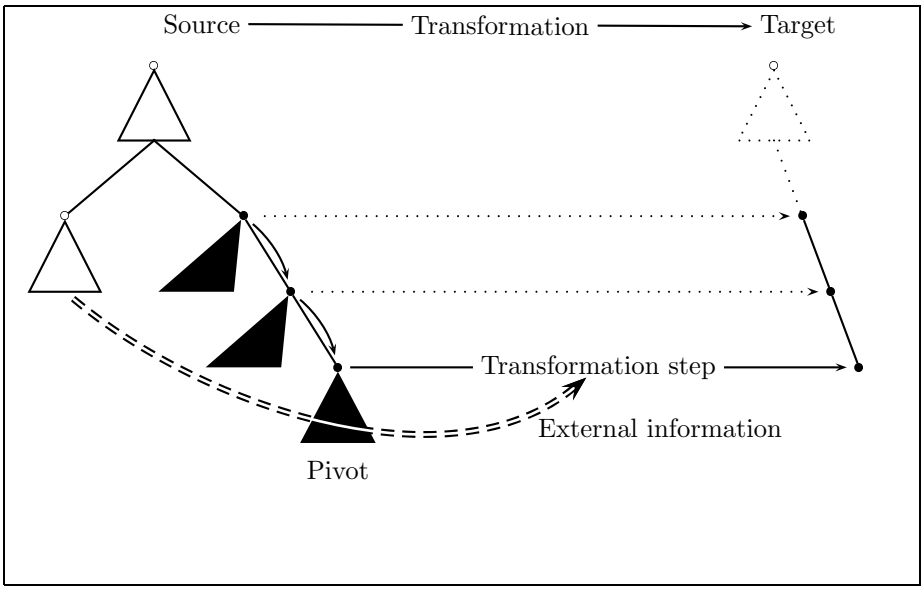

Figure 1: Schematical overview of transformation terminology

site, while the whole transformation transforms the initial source code into the optimized form where all selected methods are inlined.

The notion of scope is relevant since it has impact on the design and implementation of transformations. Typically, transformations covering only a small scope are much easier to implement than transformations with a larger scope. Partly because these small scoped transformations are conceptually less difficult, but to a large extent because tool support is often better for this type of transformation.

We make a rough characterization of types of scope. First, scope should be considered in the *source* and the *target* of the transformation. Second, in both source and target, a transformation step can have a *local* or *global* scope. To define these notions more precisely we use the term *pivot*, which was introduced by Boyle et al. [3] for the description of transformations in the *Tampr* [2, 3] transformation system. The *pivot* of a transformation is the main input element around which the transformation step revolves. In the method inlining example the method call sites form the pivots while the method body is external information for the transformation. Figure 1 illustrates these terms in a schematical overview of a transformation.

A transformation step has *local source* scope if the input consists of just the pivot or can be found in the subtree of the pivot. If input information is located outside the subtree of the pivot or no pivot can be determined the transformation step has a *global source* scope. A transformation step has a *local target* scope if there is only a single output node and it has a *global target* scope if there are multiple output nodes. Source and target need not have the same scope. Indeed all possible combinations exist in practice. The following is a summary of the combinations illustrated with examples.

Local source to Local target A source node can be directly translated to a target node. The generation of some simple documentation shown in Figure 2 is an example of this type of transformation.

1 to 1 This is a special form of the local source to local target category in which, besides being directly translated, also the locations of source and target node correspond. This means a simple traversal can be used in which every source node is translated to a target node. An example transformation is the simplification of mathematical expressions shown in Figure 3.

Global source to Local target When generating a simple target node, information from multiple source nodes is needed. The function inlining example in Figure 4 illustrates this type of transformation. In this example the method call site forms the pivot and the method definition provides extra information needed for the transformation which cannot be found inside the pivot subtree.

Local source to Global target A single source node is translated into multiple target nodes. This type of transformation is illustrated in Figure 6 with the translation of a domain-specific language to C++ code. Each variable declaration is transformed into multiple references in a target C++ file, i.e., a normal constructor, a copy constructor and an object logging method.

Global source to Global target Multiple source nodes are translated into multiple target nodes. An example transformation is function specialization illustrated in Figure 5. A function call with some constant arguments is replaced with a call to a new function that is specialized with those constant values. The function call site forms the pivot of the transformation and the function definition is external information needed for the transformation that cannot be found inside the pivot subtree. At the target side the definition of the specialized function is added and the pivot is transformed into a call to the new function.

An interesting observation at this point is the fact that the scope of a transformation is not only a technical artifact of its implementation, but is closely coupled to the problem domain. That is, scope is often inherent in the specification of a transformation. For example, inlining requires information from

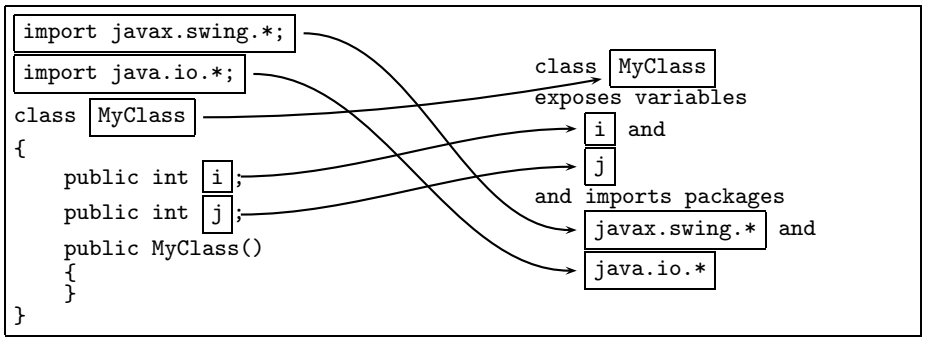

Figure 2: Documentation generation - example of a local source to local target scope

function definitions in order to transform call sites, making it an inherent global source transformation. Thus, scope is an ideal starting point when designing a program transformation. This applies to a much lesser extent to the other two mechanisms that will be discussed in this paper, i.e., direction and staging.

The scope from the problem domain can be equivalent to the scope of the transformation implementation that solves the problem, but it does not have to be. Section 5 describes several design patterns that relate the scope of a transformation in the problem domain to the implementation of a transformation.

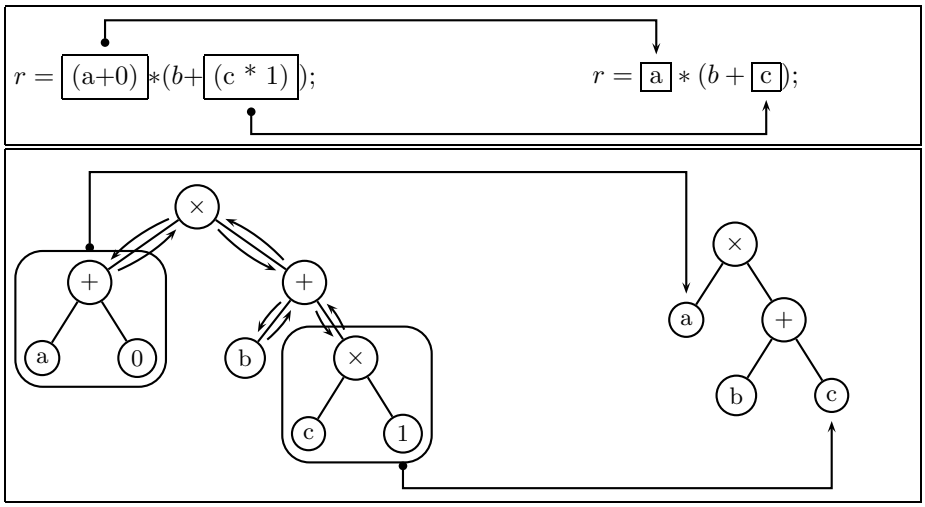

Figure 3: Expression simplification - example of a 1-to-1 scope

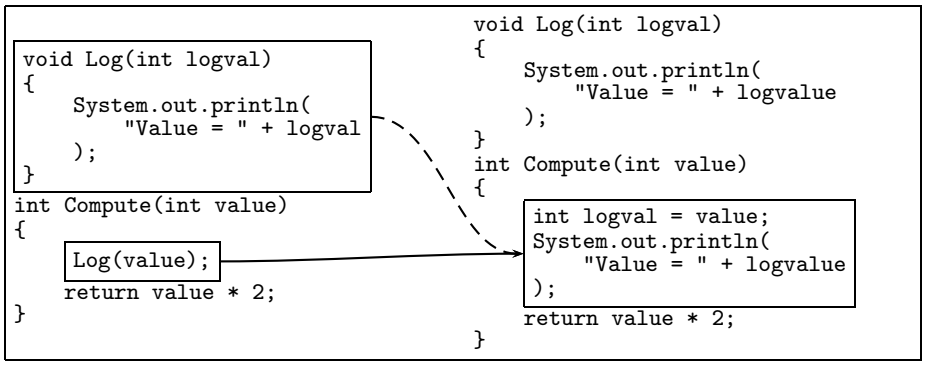

Figure 4: Function inlining - example of a global source to local target scope

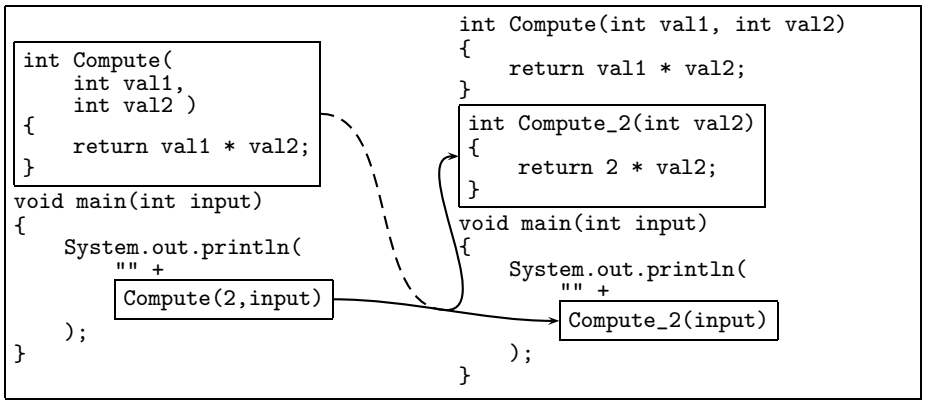

Figure 5: Function specialization - example of a global source to global target scope

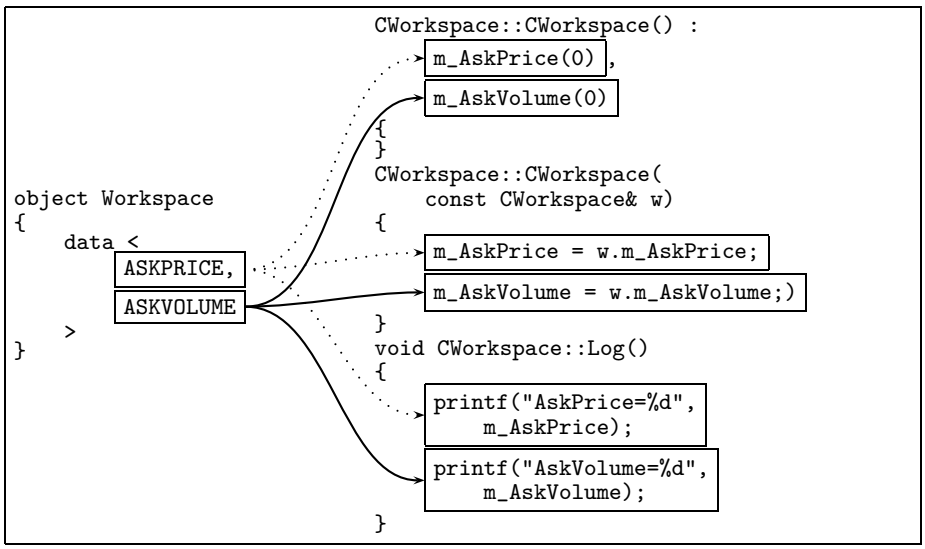

Figure 6: DSL code generation - example of a local source to global target scope

3 Direction

The notion of direction is best illustrated with an example. Compare the two code fragments of the Stratego program in Figure 7, which simplifies mathematical expressions, and the VBScript/ASP page in Figure 8, which generates a webpage. While there are evidently syntactical and implementational differences between the two code fragments there is also a fundamental difference underneath those more superficial dissimilarities. The Stratego specification defines a traversal over an abstract syntax tree applying simplification rules along the way. In this transformation the output is defined completely in terms of the input, which is why we call this Stratego example *source driven* or a *forward transformation*. The ASP page, on the other hand, exactly describes the format of the target upto a number of holes that are filled with values from the source. This is why we call an ASP page *target driven* or, because we start with the output and afterwards use the input, we also use the term *reverse transformation*.

Now we can define the notion of direction more precisely. Thus, *direction* indicates whether a transformation is controled by the source or the target. In a **forward** or **source driven** transformation the target is built by walking over the source tree and applying transformations. A schematical overview of a forward transformation is shown in Figure 3. In a **reverse** or **target driven** transformation an output template is used in combination with lookups in the source where necessary to build the target. A schematical overview of a reverse transformation is given in Figure 9.

From the examples and the definition it becomes clear why the difference

```
module ReduceExpression
imports lib
strategies
  reduce-expr = bottomup(try(ReduceAdd + ReduceMul))
rules
  ReduceAdd: Add(0,a) -> a
  ReduceAdd: Add(a,0) -> a
  ReduceMul: Mul(1,a) -> a
  ReduceMul: Mul(a,1) -> a
```

Figure 7: Source driven Stratego specification of mathematical simplification

```
<% @Language = vbscript %>
<html>
  <body>
    <h1>ASP Calculator</h1>
    The first number is <%= Request("number1") %><br>
    The second number is <%= Request("number2") %><br>
    The sum of both numbers is
      <%= CInt(Request("number1")) + CInt(Request("number2")) %>
  </body>
</html>
```

Figure 8: Target driven ASP page generating web page

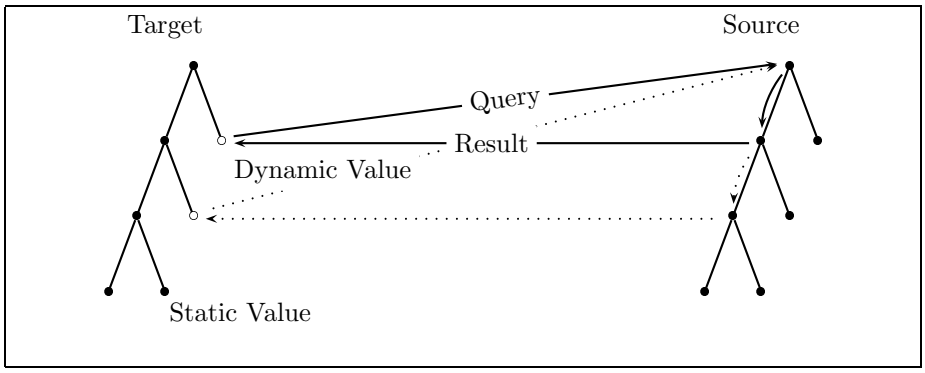

Figure 9: Schematical overview of a reverse transformation

between forward and reverse transformations is significant. In a forward transformation, traversals and transformation rules form the basis of the transformation. This implies a tool aimed at this type of transformation should have sufficiently powerful mechanisms for defining these. When using a reverse transformation on the other hand, traversals and transformation rules are of much less importance, instead we have to be able to easily lookup data anywhere in the source using some kind of query functionality.

To illustrate the difference consider using an ASP page written in the VBScript language for defining a forward transformation. As we just stated, traversals are very important in forward transformations. In object-oriented languages, tree traversals are usually implemented with the visitor design pattern [11, 31]. This is very hard in VBScript since it is not a true object-oriented language. Because of the limited generic traversal capabilities it is virtually impossible to use ASP in combination with VBScript for forward transformations.

An important prerequisite for reverse transformation is the availability of a sufficiently powerful query mechanism that makes it possible to easily lookup information anywhere in the source, based on some conditional expression. This is even more important when the input is an abstract syntax tree generated by a parser, implying that the tree format is dictated by the language grammar instead of representing the optimal data format for lookups.

While a domain study can indicate the scope of a transformation, this is less clear cut for the direction of a transformation. One observation that can be made is that reverse transformations emerged in template-based generation of web pages and programs. In this approach a large part of the target is fixed and specified by means of a template in which application-specific values are pasted. The specification of large fragments of target code requires the use of concrete object syntax [29] such that the generator looks much like the output file. For example, to allow experts of the target language to easily make small changes to the generated code without having profound knowledge of the meta-language. Besides this observation we will discuss in Section 5 a number of patterns that show how the choice for the direction of a transformation can also be influenced by other transformation aspects.

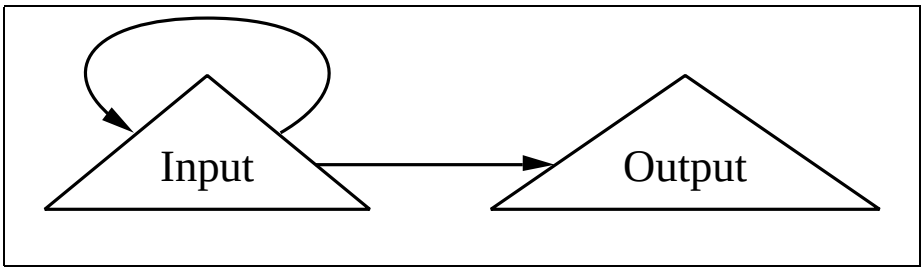

Figure 10: Schematical overview of a Multi-Stage Modify transformation

4 Staging

A final aspect of transformation mechanics is the division of a transformation in *stages*. Many transformations consist of a single traversal in which some transformation rules are applied. For some transformations, multiple traversals are used or required. For example, many optimizers traverse the abstract syntax tree multiple times, repeatedly applying optimizations during each traversal. Other transformations are also implemented in several different *stages*. We can distinguish between three different approaches when looking at the *stages* of a transformation:

Single-Stage The target is generated in one single (complex) traversal over the source. An example is the addition of log statements at the start of all method bodies in a class.

Multi-Stage Modify The target is generated incrementally by making multiple walks over the source. In every stage the same code is modified over and over again until the target is finished. See the schematical overview in Figure 10, where the arrows represent transformations. Program optimizations are typical examples of this type of transformation.

Multi-Stage Generate Every stage generates a piece of the target that is merged with all other pieces afterwards to form the final target. A schematical overview of this type of transformation can be seen in Figure 11. An example of a multi-stage generate transformation is the transformation of a DSL into a constructor, destructor and a logging method that is implemented as three distinct transformations of which the targets are merged to form the complete target class definition.

The choice for staging of a transformation can be partially influenced by domain analysis. Typically one starts with a single-stage transformation. Domain analysis can reveal that a multi-stage modify transformation is needed, for example, when a transformation has to be applied repeatedly until no further application is possible. When this is not the case a multi-stage generate transformations can be a substitute for a simple single-stage transformation, but this consideration is usually based on practical issues instead of domain analysis. This will be considered in more detail in Section 5.

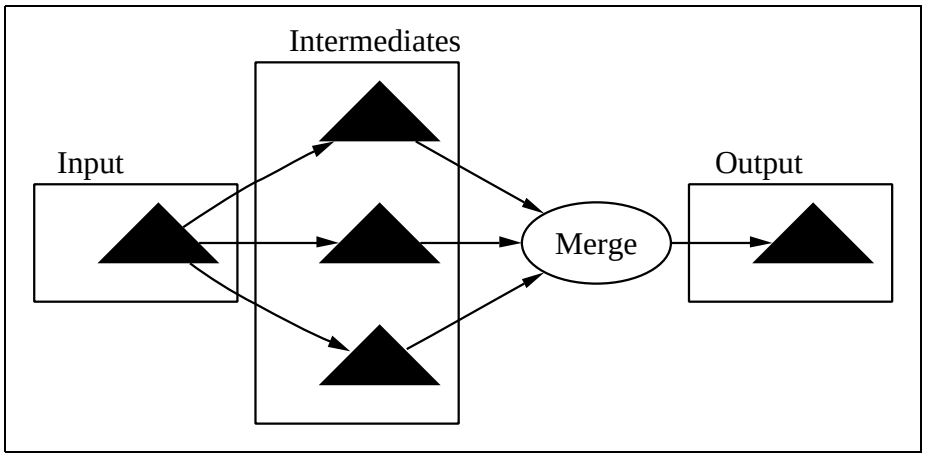

Figure 11: Schematical overview of a Multi-Stage Generate transformation

5 Design Patterns in the Transformation Landscape

In the previous sections we have discussed three fundamental aspects of transformation mechanics, distinghuishing five types of scopes, two types of directions, and three types of stages, giving rise to a landscape with thirty different types of transformations. It is evident that all three aspects can be recognized in each transformation and that almost all combinations can occur. However, some combinations make more sense than others, and tools typically only support a few of these combinations.

In this section we explore the landscape of transformation mechanisms by considering several design patterns that provide standard solutions to certain transformation problems. Although more patterns can be distinguished, we discuss the most interesting ones, which already give a good indication of how transformation mechanics can be used to describe standard designs at a high level. We also examine several concrete transformations and discuss the mechanisms that could be used in their implementation, with a focus on their scope.

5.1 Design Patterns

1-to-1 When domain analysis reveals a 1-to-1 scoped transformation is possible this will most likely be the easiest to implement using a 1-to-1 scoped single-stage forward transformation. Syntax directed translation, in which each language construct is mapped onto a combination of constructs in the target language, is a typical example of this kind of transformation.

Local-to-local The local-to-local scoped transformation is closely related to the 1-to-1 scoped transformation but an implementation will often look quite different. As an example, consider again the documentation generation in Figure 2. There are two possible implementation approaches for this example. In the first approach we use a forward transformation. First all pivots from the source are collected in a data structure that feeds a subsequent 1-to-1 scoped

single-stage forward transformation. For the example this would mean we would collect all imported packages, the classname and the public variables from the class, and then transform these nodes to the target documentation. An alternative approach uses a reverse transformation. In this approach a template for the documentation is defined, which is completed by looking up the imported packages, the classname and the public variables when necessary.

Global-to-local Transformations with a global-to-local scope can be implemented in multiple ways. If a single-stage forward transformation is chosen, global information somehow has to be acquired. This can be done using queries to retrieve data anywhere in the input tree. Another option is to traverse the whole tree collecting global information, followed by a traversal over the pivots that performs the actual transformation. A small variation on this is to first modify the tree by moving or copying some of the global data inside the pivot subtree followed by a simple local-to-local or even a 1-to-1 scoped transformation. It is also possible to implement transformations with this scope using a reverse transformation, performing multiple lookups for every target node that is to be generated. It should be noted however that if the transformation is controlled by the structure of the input a reverse transformation should probably not be considered.

Local-to-global Transformations with a local-to-global scope can be implemented in a number of ways. If we look at the example transformation in Figure 6 we can observe that in this example the generated targets are completely independent. Because of this we can simply use multiple unrelated transformations and generate each target independently. In this example that would mean we would write three transformations that each generate a single constructor or method. This is not always possible since often the generated targets are related. An example of this is the global target transformation of function specialization. If the targets are related we can make an implementation using a multi-stage generate transformation. For example, the DSL translation in Figure 6 would in this case be implemented using a transformation that for each pivot generates all constructors and methods but for just that single variable. All these methods from all pivots are later merged to form the complete target. Finally, it is of course also possible to use a more ad hoc approach. An example would be to start a traversal through the whole tree from every pivot and propagate the information necessary to create all target nodes at the right locations.

5.2 Scope of Transformations

To make the exploration of transformation mechanics more concrete, we examine a number of concrete transformations and discuss the mechanisms used in their implementation, focusing on their scope, since scope is the transformation aspect most directly related to the problem domain. Figure 12 summarizes the results of this examination. Note that some of these program transformations and their definitions were taken from [28].

Transformation × Scope	l:l	1on1	l:g	g:l	g:g
Instruction Selection		x		x	
DSL code generation			x		x
Function inlining				x	
Desugaring		x			
Constant Folding		x			
Constant Propagation				x	
Renovation				x	x
Refactoring					x
Simplification		x			
Migration	x	x	x	x	x
Reducing lang extensions	x	x	x	x	x

Figure 12: Scope of typical program transformations

5.2.1 Instruction Selection

Instruction selection deals with mapping intermediate representation (IR) trees to machine instructions. Mapping of an IR instruction to machine instructions is fairly straightforward and can be done in a simple 1-to-1 single-staged transformation. In most scenarios however this does not produce the optimal result. Instead we want to find the optimal mapping, where optimality depends on multiple factors, such as execution speed, memory usage, and program size. Sometimes a good mapping can be found by using simple optimizations to the 1-to-1 scoped transformation, such as the maximal-munch approach that prioritizes instructions that cover a large part of the IR over instructions that cover a smaller part. If however the instruction set is complex, or there exist extra constraints on the generated output, a simple 1-to-1 scoped transformation no longer suffices. Instead more contextual information is needed for deciding what mapping is optimal. In that case the needed transformation has a global-to-local scope in which the cummulative effect of transformations of single nodes is taken into account. This leads to a two-stage process in which the first stage computes all possible mappings, and the second selects the optimal one. See [4] for an overview of strategies for instruction selection.

5.2.2 DSL Code Generation

The scope of a transformation from a domain specific language to a general purpose language typically depends on how much the two languages resemble each other. In general, the source and target can be written in any type of language, and the languages can have an arbitrary resemblance. However, typically the resemblance can be assumed quite small since the DSL is usually created to provide a much higher abstraction level. Therefore, a single element from the input is typically transformed into multiple output elements, making the transformations local-to-global or even global-to-global if source analysis is required. Since these types of transformations are difficult and not very well supported among transformation tools, as we will see in the next section, the DSL language is sometimes designed with the implementation of the transformation in mind. The result is that the DSL language in such a situation will have a structure that is fairly similar to the target language, for the simple reason that it allows

for easier implementation with 1-to-1 scoped transformations.

5.2.3 Function inlining

Function inlining is an optimization that replaces a function call with the function body. This reduces the overhead of a function call at the expense of increased code size. The main benefit comes from the possibility for better optimization of the function body using information from the function call site. Function inlining is a typical example of a global-to-local transformation, since the function definition is needed in addition to the function call site, which is the pivot of the transformation.

5.2.4 Desugaring

Desugaring is a kind of normalization in which some constructs, known as *syntactic abstractions* or *syntactic sugar*, of a language are eliminated by translating them into more fundamental constructs. These new constructs are usually placed at the same location as the original ones, enabling the implementation of the desugaring as a 1-to-1 scoped transformation.

5.2.5 Constant folding

Expressions with constant operands can be evaluated at compile-time, reducing run-time and code size. This simple optimization by itself does not have great impact on most programs, since most programmers do not write much code that could be directly optimized by constant folding. However, combining this optimization with other transformations such as macro expansion, constant propagation and function inlining, makes this a very simple and effective optimization. Because it is such a simple transformation that is based only on local information, it can easily be implemented as a 1-to-1 single-stage forward transformation.

5.2.6 Constant propagation

If a variable has a constant value, i.e., it is known at compile-time, it is possible to replace all references to the variable with this constant value, reducing run-time. The main benefit is achieved when combining this transformation with constant folding, resulting in a cascading effect where large parts of a program can be evaluated at compile time. Variable usage sites form the pivot of this transformation. In order to determine if a variable can be transformed and to what constant value it should be transformed, an analysis is needed to determine if the variable was constant earlier in the program flow. This is external information to our transformation making this a global-to-local transformation.

5.2.7 Refactoring

A refactoring is a transformation that improves the design of a program while leaving its functionality unchanged [10]. Typical refactorings first require some analysis followed by making the design change which often includes changing the pivot as well as some other locations, resulting in a transformation with a global-to-global scope.

5.2.8 Renovation

Software renovation is the transformation of a program where extensional behavior is changed in order to fix an error or because the functional requirements have changed. This process practically always requires some analysis after which a pivot can be translated into one or sometimes multiple output nodes. This means software renovation involves a global source and a local or global target scope.

5.2.9 Simplification

When simplifying a program it is reduced to a normal form. Examples are the filling in of default values or algebraic simplification of expressions. This simple transformation, that looks like but is not the same as a desugaring, can usually be implemented using a 1-to-1 scoped transformation.

5.2.10 Migration

When migrating a program it is transformed to another language at the same level of abstraction. These migrations can often only be automated successfully if the languages are fairly similar. When this is the case most of the transformation can be done using a simple 1-to-1 scoped transformation. In most non-trivial migrations however there are also some structures that cannot be directly translated. To transform these often some global analysis is needed and/or some code has to generated at multiple locations. Therefore nontrivial migrations often involves a mix of all types of transformation scopes.

5.2.11 Reducing language extensions

This transformation deals with reducing arbitrary language extensions, from a domain specific embedded language (DSEL) into normal host language constructs. This transformation shows similarities with the desugaring transformation in the sense that both reduce certain statements to more basic statements, but there is one big difference. While a desugaring step is very small, which means a certain expression is desugared to one or maybe a few other expressions that are inserted at the location of the original statement, a language extension usually has a much higher level of abstraction resulting in a more complex transformation. It is very well possible a single extension statement will be reduced to multiple code statements at different locations. For example, inserting variable declarations for auxiliary variables used by the translation of the extension. Furthermore, it is very well possible some intelligent analysis is needed before the reduction can be done. We can therefore conclude that all types of transformation scopes could be needed when implementing this type of transformation.

6 Tool Survey

In this section we present a systematic technical evaluation of transformation paradigms and tools aiming at discovering the transformation mechanics supported by a tool. For obvious reasons it was not possible to review all existing

transformation solutions, so we selected eleven of these solutions trying to cover many fundamentally different tools and paradigms. Although we use the term tool, the survey is not limited to classical program transformation tools, but also includes some general purpose languages to provide a broad perspective on all solutions.

We should stress that we do not aim at finding the best transformation tool currently available. The fact that a tool lacks certain functionality does not automatically mean it is inferior to a tool with more functionality. Limited functionality dedicated to support a restricted problem domain can make a tool more straightforward to use than a more generic solution. It completely depends on the user and the problem that needs to be solved what works best. Aspects such as the syntax of a language, the user interface of a tool, and the learning curve should also be considered when evaluating a tool for a specific purpose or project. Evaluation of these aspects is beyond the scope of this paper.

Because we have only firsthand experience with a limited number of tools, we base the evaluation on tool descriptions in the literature. Although we have spent a considerable amount of time studying papers and documentation it is very well possible we erroneously report some features to be missing for some tools just because we were not able to find references to these features in the literature. This can be regarded a weak point of this evaluation, but it also emphasizes the importance of a common idiom for describing the transformation mechanics in a systematic way to ease tool comparison.

6.1 Tools and Functionality

Recognition of transformation mechanics in a tool is difficult without systematic documentation or firsthand experience. Therefore, we have compiled a short list of transformation features that play a role in the implementation of mechanics and which are easier to recognize in tool descriptions: object-syntax, traversals, queries, and data acquisition. First we introduce each of the tools and discuss their support for these features. Then we determine the support for transformation mechanics of each tool.

Object Syntax Meta-programs analyze, transform and generate object programs. When matching or building object program fragments one can use abstract syntax (AS) or concrete syntax (CS). Especially for writing large object fragments concrete syntax is preferred, while for smaller transformations abstract syntax is usually preferred because it is unambiguous and does not require special parser support.

Traversals For non-trivial transformations it is often necessary to traverse the input tree. Many different approaches exist, some more powerful than others. Especially important is the capability for *generically* specifying traversals, i.e., without having to spell out the traversal for each data type and each transformation.

Queries It is often useful to retrieve data from an arbitrary location in the input. A common approach is to write a query to exactly select some input

Tools × Functionality	CS	AS	Traversals	Queries	Data Acquisition
OO Languages		x	+	−	+
Functional Languages		x	−	−	−
String based generators	x		−	+	+
OpenC++/OpenJava		x	+	+	+
JTS/Jak	x	x	+	+	++
Alchemist		x	−	−−	−−
TXL	x	x	−	−	−
ASF+SDF	x	x	+	−	++
Attribute grammars		x	+	−	++
XSLT		x	+	++	++
Stratego	x	x	++	+	++

Figure 13: Evaluation of tool functionality

elements based on structure and values. Here also support varies greatly between tools.

Data acquisition A transformation often requires data from multiple locations in the input, so before a transformation can be performed this data has to be acquired. Although this feature might resemble much the queries feature, it is in fact different. Queries can be useful for data acquisition, but a more popular approach is the usage of some kind of traversal that collects interesting values. The support for this important feature however varies from tool to tool. Since both queries and traversals influence the ease with which we can acquire data the score will be influenced by the score we assigned to those two features.

Summary Table 13 summarizes the support for transformation functionality for the tools we have examined. In the first two columns we only check whether or not support is provided. On most other columns however we also show how well certain features are supported. Because it is not easily possible to give an exact score we decided to reward a score that has four possible values: ++ if the support is excellent, + if there is some support, − if support is weak and −− if there is no support at all.

6.1.1 Object-oriented languages

Object-oriented programming is a popular programming paradigm. In particular, the C++ and Java languages are widely used. Because of this popularity it is interesting to see how well this type of programming language performs when using it for implementing program transformations.

When we look at the input we typically see that the complete AST is available in the form of a hierarchically structured collection of objects. These objects are created from classes that correspond to the non-terminals in the grammar. These classes must be handwritten, although sometimes a tool can be used to generate these from a grammar definition. These general purpose languages typically do not have any support for working with concrete object syntax or custom language extensions, which makes working with the AST quite verbose

when for example a new tree fragment must be built or matching against a pattern has to be done.

Traversals are available in the form of the visitor design pattern [11] which allows for easy tree traversals. However, one of the problems of the visitor design pattern is the rather limited traversal control. Visitor combinators fix this problem, but they are only recently explored [31]. They look promising, but because a lot of code has to be written, additional tools must be relied upon to generate this code. Because it is not yet clear how well this approach will work in real scenarios and also because it is not an out-of-the-box solution it will not be considered here.

There is also no real query functionality available. Simple queries are possible using standard methods for getting the parent and children. This however often requires a lot of type checking and casting which clutters code. Also no special pattern matching support exist which makes matching of simple tree structures very verbose. This basically makes only simple local matching manageable. Simple collect queries can be mimicked by visitor but if collecting has to be a bit more intelligent visitors fail because of the weak traversal control and matching support. Visitors are very good at collecting data during traversal which makes data acquisition potentially powerful. Due to a lack of pattern matching and query support however this support is suboptimal.

6.1.2 Functional languages

Especially in research projects general purpose functional programming languages like Haskell and ML are very popular and there have already been some published results applying this paradigm to program transformations. Because there are many very different functional languages it is difficult to generalize over them for this evaluation. That is why we will assume the functional language Haskell where necessary. The reason for choosing Haskell is because it is a modern language which is used a lot in recent research.

In a functional programming language every language construct is represented by a special constructor. Just as with object-oriented languages these constructors must be handwritten or a tool must be used to generate these from a grammar definition. Just as the other general purpose solutions most functional programming languages do not have support for concrete syntax. It should be noted however operations on the AST can often be defined in an elegant way solving some of the problems of writing transformations in abstract syntax.

Traversals often have to be written by hand. This means that for every node in the AST you have to define how the traversal should recurse into the children. It should be clear this severely limits usability. This problem is recognized and gradually several solutions [17, 15] are presented that will allow for generic traversals. Although they look promising, these solutions require extensions to the compiler or the language. Therefore, we will not consider these solutions in this overview.

There is no special query support. However, some languages, including Haskell, have very nice pattern matching support. The only problem is that because of the limited traversal functionality this is only useful for matching against patterns close to the current location. Furthermore, the root of the input tree is not easily available, only allowing for queries in the current subtree.

We can conclude that only very simple queries can be implemented and that more complex queries are not supported.

6.1.3 String-based Generators

There exist quite a few string based program transformation tools, mainly because they are very simple to develop using existing tools and environments. Examples are ASP and JSP that were intended for the generation of dynamic webpages but similar tools have also been employed for meta-programming. The Jostraca [21] tool is an example of this. These tools are characterized by the fact that the output is specified in the form of plain strings. The reason for including them in this overview is because they provide a very easy to use approach to program transformations.

A characteristic feature of string based generators is that the target template can be defined in concrete syntax. This concrete syntax is not parsed, however, but is treated as plain strings. As a result the AST is never available. Since the engine does not care about the content of the strings any output can be generated, ranging from HTML to C++ code.

These engines typically get their input from a flat input structure like page arguments or a database through the usage of SQL-queries. Because of this traversal code is underdeveloped. If the input or output is a tree structure, depending on the meta-language, the traversal can sometimes be handwritten using the visitor design pattern. Two very popular meta-languages, VBScript and PHP however do not support this.

There is no real query support and basically you can only access global variables. Because of the flat input structure however, this 'query' functionality is sufficient for typical applications of string-based generators. Because of this sufficient query functionality we rated data acquisition support also as sufficient. Again here also we took into consideration the fact that the typical input of these engines is flat, allowing only for simple traversals and as a result data acquisition support does not have to be very advanced.

6.1.4 OpenC++/OpenJava

OpenC++ and OpenJava provide an interesting approach that extends an object-oriented language with meta objects that control the compilation of the class a meta object is assigned to. These meta objects can be used to make modifications to a class, for example to improve efficiency but also to desugar simple language extensions. OpenC++ [5] is interesting specifically because it is one of the earlier approaches in this direction and recently the same approach was applied to the Java language [23]. Both tools operate directly on the AST. OpenJava provides a little more abstraction compared to OpenC++ by providing an extra interface that hides some of the irrelevant details of the AST, like the declaration order of some variable modifiers. Neither tool provides any support for concrete syntax.

The output of the two tools is C++ and Java respectively and cannot easily be changed. The input language is the same language with some extensions. There are some fixed extensions that support the attachment of a meta object to a class. Furthermore, there is support for simple language extensions in the

form of custom keywords at a several fixed locations, e.g., in front of a class or variable declaration.

When a class is transformed an implicit traversal is performed over the input, which calls the right meta object method for a number of fixed elements such as method call, variable get, variable set, etc. Through these method calls the target code is modified. Although this built-in traversal is sufficient for simple code generation, more advanced traversals may be required for other transformations, which seems to be impossible.

Querying data can be done through a large number of available methods very similar to Java or C# introspection. A meta class, for example, has methods for getting all implemented interfaces, all class variables and all methods of a class. For example, the method descriptors provide easy access to information such as the return type, arguments, the body, etc. A more complex query, such as getting all referenced variables inside methods that start with 'log', however, becomes very verbose. One of the reasons is the lack of generic traversal functionality that forces the complete specification of every traversal step.

Data acquisition could have been better if query functionality would be more powerful, but the readily available introspection functionality is quite complete considering the targeted transformations which makes data acquisition support quite powerful.

6.1.5 JTS/Jak

Jak is part of the Jakarta Tool Suite [1] and provides an extensible Java language that supports meta-programming. One of its special features is that it allows for arbitrary language extensions and that concrete syntax can be used for meta-programming. The JTS tool is based on the GenVoca model allowing for easy combining of different components.

Jak provides an extension to Java to make it a meta language for generating other Java programs. Although it is possible to write transformations using abstract syntax there is an extra extension that includes functionality to write concrete syntax that is later reduced to abstract syntax. The current JTS implementation only seems to support Java output, although it should be relatively straightforward to extend this to other languages.

Traversals are possible through cursors that define a location in the tree. Although this approach allows for easy traversals through a whole tree, node-specific actions can be very verbose requiring a lot of type testing using the `instanceof` operator and casting. Since many intelligent traversals rely on node-specific, they can be awkward to implement. There is however a special matching mechanism available that is implemented through a simple language extension and allows easy iteratation over nodes from the input tree matching an arbitrary pattern. Besides looking up data, this matching support can also be easily used to make changes to the tree at specific locations. This can therefore also be used for simple traversals. One of the primary applications of the matching support is queries, since it allows for easy data lookups that match against a certain pattern anywhere in a subtree. Quite complex queries can be made using this matching support, although they probably require multiple of these matching patterns to be applied successively, which can be quite verbose.

The `reduce2java` method that is responsible for the actual transformation of a node into Java code has a properties argument, which basically is a hashtable

that allows passing around arbitrary information. Although no usage examples could be found in the literature, we expect this feature to provide quite good data acquisition support. The only downside is that it only works in the last transformation stage, just before the final Java code is generated. For custom traversals this support is not available and has to be implemented manually. Another option is presented by the `SymbolTable` mixin, which presents meta information about packages, classes, methods and so on through a globally available interface. Although unfortunately it is not possible to use the matching support on AST nodes that are stored in the symbol table, the matching support can of course be employed to run simple queries against the root of the AST to acquire data. If we sum up all these features we can conclude JTS provides good support for data acquisition.

6.1.6 Alchemist

This approach described in [18] is primarily of interest because it was one of the first program transformation environments and because it is based on TT-grammars [13], a paradigm not commonly used but that looks interesting because it has been intended exactly for the definition of program transformations.

Transformations in Alchemist are based on grammar productions. The tool allows specification of the transformation of a grammar production of the source into the production of the target. Transformations are defined completely in terms of grammar productions, which actually is a mix of concrete and abstract syntax but in the context of this evaluation we consider it abstract syntax. After specifying the exact transformation the system will perform a simple traversal over the input to transform every node. This traversal is implicit and cannot be customized in any way. Furthermore, it is not possible to add user-defined traversals over the input. Support for data acquisition and queries is absent, which is recognized by the developers.

6.1.7 TXL

The tree transformation language TXL [6] is a language designed for specifying rule-based source-to-source transformations, claiming to be a general purpose transformation system. Due to its long history – work on a direct predecessor started as early as 1990 – this tool has been used in many different industrial projects.

In the transformation rules a mix of concrete and abstract syntax can be used. From the literature it does not become clear how the parsing of this concrete syntax works. From the different applications shown in different papers we assume it is relatively easy to generate code for an arbitrary language. The TXL engine applies all transformation rules repeatedly in a nondeterministic way until no more rules succeed. When a rule is applied it is possible to call other rules that operate on a supplied argument that could for example be a child node. This provides a basic mechanism of custom traversals over the input tree, but it is not very powerful. There is no query support, but template matching support is quite powerful. It is however unclear if and how this functionality allows one to retrieve input from an arbitrary location in the input.

Support for data acquisition is rather limited. If information from elsewhere is needed in a transformation one option is to first do a custom traversal that

collects the necessary data followed by a traversal that takes all collected values as arguments and traverses to the pivots. Due to the weak traversal support and due to the bad scaling characteristics of this approach this is not a very good solution. A possibly better approach, that is also aimed at allowing for transformations in which source and target language are different, is to use several intermediate grammars and apply multiple successive rewrites from one grammar to the next. This way global source transformations become possible, but the code becomes hard to maintain, because of two reasons. First, the intermediate grammars do not have any real world meaning and can therefore become hard to understand. Furthermore, if the number of intermediate grammars becomes large, there is quite a high risk that a lot if not all of those grammars will have to be modified if new functionality is needed. It is reported in [8] that a COBOL to Java transformation required about seventy of these intermediate grammars. A solution for this problem is the usage of a *union grammar* which means the source and target grammars are combined in such a way that all intermediate representations can be expressed in that single grammar. This indeed solves the problems caused by having many intermediate grammars, but it remains unclear whether a suitable grammar can be constructed for any two source and target grammars, especially if they are very different.

6.1.8 ASF+SDF

The ASF+SDF meta environment [9] provides a complete environment for program transformations based on the term rewriting paradigm. One of its qualities is that it provides strong type checking on transformation rules. Another reason that make it an interesting tool to study is because it has been used in a number of industrial projects.

The equations that define the transformation are written in concrete syntax. This concrete syntax is fully parsed and the resulting AST is used internally. Besides the equations, a transformation specification also consists of a complete grammar definition of both input and output, which makes it possible to easily generate any type of output.

Traversals are fairly well developed. There are two basic traversals, bottomup and topdown. It is also possible to define new traversals by manually specifying inside every rule how a transformation should recurse into its children. Due to the tight coupling of the transformation and this type of traversals it is unfortunately not possible to separate this functionality, making reuse of custom traversals very difficult.

There is no special query functionality that allows for easy lookups of information. Although the rich equation functionality can be used for expressing certain queries, it does not provide a very compact syntax and is only practical for querying the current subtree not allowing for easy access to other parts of the input tree.

There are a number of special features that allow for easy data acquisition. First of all it is possible to collect data in extra attributes during a manual traversal. This suffers from the problem that the addition of a single attribute will affect many equations, which presents a scaling problem. The usage of the predefined traversal types accumulator or transformer provide a much nicer approach. For these predefined traversal types the system generates default rules that will automatically handle the traversal over terms for which no traversal is

specified. It even takes extra arguments into account, making data acquisition easy. This however only works nicely if the traversal can be captured in a basic bottomup or topdown traversal. If this is not the case there is the option of overriding the traversal at some key points, but these overrides should specify exactly all extra arguments used, introducing again the same scaling problem as in the manual traversal.

6.1.9 Attribute grammars

Although attribute grammars have a long history [14] they have never managed to become widely used. Recently however there is some renewed interest in them [22, 12]. Although there is hardly any experience using attribute grammars for program transformations we included it in this evaluation because it is yet another programming paradigm and it looks like it can provide an attractive implementation of global source transformations worth studying.

The AST forms the basis for attribute grammars since attributes are assigned to every node in the AST. For each of these attributes a specification defines how its value can be computed based on values from the children, parent or other attributes. The output of an attribute grammar can be anything ranging from a simple value, such as the number of variables used, to much more complex values, such as a transformed input tree. However, since attribute grammars are hardly ever used for building new trees they usually do not support concrete syntax for this.

Traversals are special in the context of attribute grammars because they only exist implicitly. When evaluating an attribute grammar the framework or tool will determine the order of attribute calculation, thus allowing for some very intelligent 'traversals'. In some situations custom traversals might be needed which are often not sufficiently supported.

The attribute grammar paradigm does not provide special support for queries. Although attributes can also quite easily be used to make data available elsewhere in the tree, queries are more lightweight and allow for easy ad hoc value retrieval, while attributes are a bit more heavyweight in the sense that a system-wide set of attributes should be defined that should be solved by the framework. If the attribute grammar system is implemented inside an existing general purpose language query support from that language can be used. Unfortunately, many general purpose languages have weak query support so this is not of much help.

Gathering data from anywhere in the input tree is exactly what attributes do best. Just define which value should be put in attributes and the framework will make sure all referenced values will be transferred to the right locations. Although this sounds very neat, there are some catches. The biggest catch is that most attribute grammar systems do not have a type of attribute that can be directly used for this, but instead they work with synthesized and inherited attributes. To enable sharing of information through the whole tree we would first write a synthesized attribute that collects information. The value of this synthesized attribute will be assigned to an inherited attribute at the root of the tree that distributes this information through the whole tree making it available anywhere.

Although the explained approach is rather straightforward it requires at least two attributes for looking up one simple external value of our transforma-

tion. For all of these attributes simple copy rules are needed, which number can quickly explode if a number of global source transformations with widespread information are to be implemented. It is therefore of vital importance the used framework has powerful support for the definition of default copy rules. Furthermore it is important support is available to group together related attribute definitions to be able to handle the large amounts of attributes this approach will need. We conclude that the attribute grammar paradigm has excellent support for data acquisition, but it should be noted that there can be implementational issues that might limit the usability.

6.1.10 XSLT

The very popular functional transformation language XSLT [33] can be used to transform XML documents into arbitrary output and cannot be overlooked when studying program transformations, although up to now it has not been used much for program transformation.

A typical XSLT rewrite rule, or *template* in XSLT terms, receives as input an XML document, for example, the abstract syntax for a Java file. The right hand side of these templates is the output also in terms of XML. However, since simple string values can be used as XML values, concrete syntax can be used just as well. Although this approach is widely used, we will not assume this approach since it does not provide any format checking and suffers from escaping problems. Another reason is that we think XSLT was not designed to be used like this, which is supported by for example the new XSLT 2.0 specification that allows for applying a transformation to the result of another transformation. This is of course only possible if all transformation results are specified in terms of XML. Although we think only XML should be output, this XML can of course be abstract syntax for any language and can be pretty printed to the concrete syntax of that language.

It is possible to define traversals by manually recursing into the children of a node, but this does not allow for a clean separation of traversal and rewriting. For this reason, intelligent traversals are hard to write and maintain. This is compensated by the powerful XPath matching functionality [32], which can be used for defining the nodes a template should be applied on. It allows not only for matching against a specific node, but also against the context a node occurs in, for example by putting restrictions on it's ancestors. Query support, through XPath expressions, is very well developed. These expressions allow selection of nodes precisely, using a compact and easy to learn syntax. Furthermore, it always allows direct access to the whole tree and not only to just the current subtree. Data acquisition is easy, primarily because of the powerful queries. Furthermore, it is also possible to manually pass extra arguments to recursive template applications allowing propagation of data through a traversal. Finally, there are variables inside templates and global variables that can hold arbitrary values, making sharing of data very easy.

6.1.11 Stratego

The special purpose transformation language Stratego is based on term rewriting [30] and can be seen as the first incarnation of the strategic programming paradigm [16]. It provides numerous features such as hygienic variables, concrete

object syntax and dynamic rules that make it very suitable for implementing program transformations.

Internally all transformations operate on an AST. The programmer has the choice between abstract syntax, usually preferable for small transformations or small code fragments that are ambiguous in concrete syntax, or concrete syntax, that can be mixed into a Stratego program using a fully customizable syntax for defining quotation and antiquotation [29]. The output of a transformation always is an AST that can be pretty-printed to concrete syntax. Since the programmer is completely free to choose the format of the output AST any type of output can be generated.

Because of the strategic programming paradigm used in Stratego traversal support is excellent. Generic term traversal is readily available through the usage of congruence operators and generic basic traversal strategies such as `all` and `one`. These can be combined using strategy combinators for sequential composition, non-deterministic choice and many others, to build very complex traversals. In fact, Stratego includes a library with many common generic traversal strategies such as topdown, buttomup, collect, innermost and many others which is evidence of the power of this mechanism.

There is no standard query functionality available, but powerful pattern matching support in combination with intelligent traversals provides access to data. Although this approach at first sight only seems to give access to data in the subtree of the input of a transformation, scoped dynamic rewrite rules [27] can be used to access the initial input tree at any time during the transformation. Furthermore, a recent study [26] shows the user-defined extension with XPath-like [32] queries using a custom, application-specific, syntax.

Data acquisition is possible in a number of ways; scoped dynamic rewrite rules provide a very interesting approach of making data available elsewhere in a way nicely fitting in the term rewriting paradigm. Furthermore, it is possible to tuple the normal input tree with other data and carry along this information during the traversal. This can present a scaling problem, since addition of new arguments will likely force modificaton of many rules. An approach that scales much better is the definition of a special traversal that holds an environment which values can be used inside every transformation. The standard Stratego library already contains a number of these traversal strategies such as `env-topdown` and `env-bottomup`.

6.2 Tools and Mechanics

Based on the previous analysis of tool functionality, we can now derive an analysis of the support for transformation mechanics. Figure 15 presents a table summarizing the results of this analysis. The rest of this section justifies the results for each tool. For this purpose, Figure 14 relates transformation mechanics to the tool features discussed in the previous section. That is, for each of the mechanical aspects introduced before, the figure indicates which tool features are required for its implementation. Also, in parentheses we introduce an abbreviation used as reference in the table. Note that only a simplified view of the relations between mechanics and features is given since much of this information can also be derived from the design patterns section. Furthermore, note that the scores for the different mechanisms do not only depend on the features introduced earlier, but also take tool-specific features and limitations

- Scope
  - *1:1 (1on1)* transformations depend mainly on traversals.
  - *Local-to-local (l:l)* transformations depend on powerful traversal support and support for reverse transformations. Also good data acquisition support can be helpful.
  - *Local-to-global (l:g)* transformations can implemented with multi-stage generate transformations as can be concluded from the design patterns in Section 5.
  - *Global-to-local (g:l)* transformations can be achieved by collecting all data prior to transformation, requiring support for queries or data acquisition. In some case reverse transformations can be used.
  - *Global-to-global (g:g)* transformations start with good data acquisition support. Multi-stage generate transformations can be used sometimes. Mostly application-specific solutions must be chosen.
- Direction
  - *Forward (fwd)* transformations require good traversal support.
  - *Reverse (rev)* transformations require queries for retrieval of data from arbitrary locations. Often large target fragments are specified, which greatly benefits from concrete syntax for readability.
- Stages
  - *Single-Stage (ss)* transformations are the basis that is always supported.
  - *Multi-Stage Modify (msm)* transformations require an abstract syntax tree for intermediate representation allowing analysis and modification.
  - *Multi-Stage Generate (msg)* transformations also require abstract syntax trees, although some limited merging is also possible if no tree is available.

Figure 14: Relation between transformation mechanics and tool features

into account. The following subsections provide some explanation about the assessment of each tool. Due to space limitations a full explanation for each score for each tool cannot be given, but only the most interesting scores are discussed.

6.2.1 Object-oriented languages

The visitor design pattern can be used for forward transformations using the fairly good traversal support provided by the visitor design pattern. The support for reverse transformation on the other hand is not very good, particularly, since concrete syntax is not available and due to weak query functionality. Visitors provide reasonable traversal support, which makes 1:1 scoped transformations

Tools × Mech.	fwd	rev	ss	msm	msg	l:l	1on1	l:g	g:l	g:g
OO Languages	+	−	++	++	++	+	++	+	+	+
Functional Languages	+	−	++	++	++	+	+	−	−	−
String-based generators	−	++	++	−−	−−	++	−	−	++	−
OpenC++/ OpenJava	+	−	++	++	−−	+	+	+	+	+
JTS/Jak	+	+	++	++	++	+	++	+	++	+
Alchemist	+	−−	++	−−	−−	+	++	−−	−−	−−
TXL	+	−−	++	++	−	+	++	−	−	−
ASF+SDF	++	−−	++	++	++	++	++	+	++	+
Attribute grammars	++	−−	++	−	−	+	++	+	++	+
XSLT	++	++	++	−−	−−	+	++	+	++	+
Stratego	++	+	++	++	++	++	++	++	++	++

Figure 15: Tool support for transformation mechanics

relatively easy to implement. Global source transformations are also reasonably well supported because data acquisition is fairly good.

Global target transformations are also supported. If the targets are relatively close to the pivot you can exploit the fact that object-oriented languages are imperative and allow you to directly modify the AST at the right location after a short and simple traversal using calls like `getParent()` or `getChildren()`. If the distance is larger this type of traversal becomes problematic so you might have to resort to another approach: first do a collect of all needed information after which the transformations at all target locations can take place. This however, for reasons discussed earlier, is not a perfect solution.

We can summarize that a plain object-oriented language can be used for numerous program transformations, but compared to other evaluated tools it simply lacks some specialized features. As a result almost all types of transformations are supported, but this support is never very good.

6.2.2 Functional Languages

Support for forward transformations is relatively good, although the lack of traversal support limits usability. Reverse transformations are difficult because concrete syntax is not supported and because queries are not very powerful. Due to limited traversal control, queries and data acquisition support combined with the nature of functional programming languages that disallows you to easily make non-local changes to the tree while traversing it, only 1-to-1 and local-to-local scoped transformations are easy to implement. Although there is some form of data acquisition that allows global source transformations, implementing this type of transformation often turns out to be difficult. It is very easy to successively apply transformations to the input tree allowing for multi-stage transformations.

6.2.3 String-Based Generators

String-based generators are completely centered around reverse transformations and provide good support for that. Forward transformations are often much more difficult to implement because of the limited traversal support. Multi-stage transformations are very difficult if not impossible, because the AST is not available. Another practical problem that hinders this type of transformations is that most engines automatically write all output directly to the receiver making multi-stage transformations impossible. The only two transformation scopes that are very well supported are transformations with a global-to-local or local-to-local. All other transformations suffer from the weak traversal and multi-stage support.

6.2.4 Attribute Grammars

By design, attribute grammars are very suitable for defining forward transformations. On the other hand it is unclear what reverse transformations would look like in the attribute grammar paradigm. Support for these is therefore nonexistent.

The 1-to-1 scoped transformations fit in these frameworks and can be implemented intuitively. The real value of attribute grammars lies in their use for global source transformations, because data acquisition support is good. Global target transformations are a bit more difficult to implement due to a lack of special support for this type of transformations. In imperative implementations it is possible to make direct changes to non-local tree parts, although it is questionable how well an attribute grammar can cope with this type of changes to the tree while evaluating the attributes. Also this calls for some custom traversal support for which support is not very good.

As discussed earlier multi-stage generate transformations can sometimes also be used for global target transformations, but this can also prove difficult since it is unclear how attribute grammars can be used to merge the multiple generated trees. A general-purpose host language however could provide good support for this type of transformation. Multi-stage modify transformations present another problem. Since often a tree with attributes cannot be easily changed without breaking attribute evaluation, the new tree should be generated as an attribute value. Combining this with multi-stage modify transformations means that the new tree itself should again be an attribute grammar. Although this does not seem to present any fundamental problems, hardly any transformation framework provides documented support for this type of recursive attribute grammars.

6.2.5 OpenC++/OpenJava

The main implicit traversal in OpenC++ and OpenJava has a forward direction, traversing over some common language elements such as method calls, variable getting and variable setting. There is no support for reverse transformations. Simple 1:1 scoped transformations of these nodes are very easy to implement, transformation of other nodes however is problematic, because it is not possible to define a custom traversal over these nodes.

Both tools do provide quite nice query support that allows easy lookup of information from the input tree, but in general they have the same strengths and

weaknesses as normal object-oriented programming. For sure these tools add an extra layer over normal object-oriented programming helping you to define certain transformations, but it has been specialized to a large extent to reducing simple language extensions while little attention has been paid to other types of transformations.

6.2.6 JTS/Jak

The fairly good traversal support combined with the availability of concrete syntax and query support allows for both forward and reverse transformations. Both could be supported better. Forward transformations suffer from the less than optimal traversal support, while reverse transformations could benefit from a better antiquotation mechanism. It seems that JTS only supports easy insertion of meta-variables, but that it does not support a loop containing quotations. Although it still is an improvement over normal Java, the language extended by Jak, this does limit usability of reverse transformations. The score for the different scopes is quite similar to object-oriented languages. The only clear exception is support for global-to-local transformations, which clearly benefit from the good data acquisition support combined with nice reverse transformation and query support.

Furthermore, the system has some other small improvements such as the availability of a symbol table that slightly improves query support and allows for easy looking up of values. Features such as this certainly increase usability, but it does not present some fundamental improvements over normal object-oriented approach, therefore not justifying higher scores on for example the other transformation scopes.

6.2.7 Alchemist

Although tree transformation grammars can be very good at 1-to-1 and some local-to-local forward transformations, there is absolutely no support for traversals, queries or data acquisition. This makes all other transformation scopes very hard to implement. It is unclear what a reverse transformation should look like in the context of tree transformation grammars. It is therefore no surprise that Alchemist does not support this. Multi-stage transformations are possible in theory, but no references have been found in the literature on this functionality and we assume it is not present.

6.2.8 TXL

The TXL tool is completely aimed at 1:1 scoped forward transformations and indeed these are easy to implement. Also since very simple traversals are possible, some simple local-to-local transformations are possible. However, because of lacking support for reverse transformations combined with weak traversal support and lack of powerful data acquisition functionality, both global source and global target transformations are very hard to implement.

6.2.9 ASF+SDF

The ASF+SDF Meta-Environment is based on forward transformations. Although concrete as well as abstract syntax is supported, some basic limitations

prevent the implementation of reverse transformations. The easiest type of forward transformations is the 1-to-1 scoped transformation for which excellent support exists. Local-to-local transformations also have excellent support due to good traversal and data acquisition support. Global source transformations are also easy to implement due to this data acquisition functionality. Implementing global target transformations is less straightforward since no special functionality exists to make these possible. Also reverse transformations cannot be used for this type of transformations, because these are not available. Fortunately, this tool can handle multi-stage transformations easily that can be used for certain global target transformations. Furthermore, traversal support is quite good also allowing for some ad hoc solutions using multiple traversals.

6.2.10 XSLT

This tool has good support for both forward and reverse transformations. Although for other tools we used concrete syntax as one of the criteria for good reverse transformations, we do not do so for XSLT because basically the XML inside XSLT is used as if it was a concrete syntax. This does however not mean XSLT does not suffer from the problems other tools have with reverse transformations in combination with abstract syntax.

It would for example not be a very good idea to try to generate large pieces of C++ code using reverse transformations and plain XML inside XSLT. Instead you would have to use a tool that parses concrete C++ syntax fragments during compilation and transforms it into an XML AST after which the XSLT transformation can be used as usual. However, this is hardly used, probably because XSLT is typically used for making small transformations on XML documents instead of some higher-level language. Despite this issue, we rewarded XSLT the maximum score for reverse transformations because commands like for-each combined with very powerful XPath query support do provide the best possible reverse support given the limitations associated with abstract syntax.

Good traversal support makes 1:1 transformations easy. This combined with the available data acquisition functionality also allows for powerful global source transformations. Global target transformations are slightly less easy to implement, because XSLT does not provide any special functionality for this type of transformations. Furthermore, multi-stage transformations are impossible since XSLT cannot reapply transformations on the result of another transformation and because of this limitation multi-stage generate transformations cannot be used to implement global target transformations. This severe limitation will be cured in the upcoming XSLT 2.0 specification.

6.2.11 Stratego

The excellent traversal functionality of Stratego allows for easy forward transformations. Reverse transformations are also possible using abstract and concrete syntax. The concrete syntax approach has powerful customizable quotation and anti-quotation support allowing for perfect mixing of meta and object language. Even when using concrete syntax the AST is always available. Therefore, multi-stage transformations are fully supported. Furthermore, Stratego supports the creation of multiple trees and apply transformations on any of these (temporary) trees, thus supporting multi-stage generate transformations.

Because all these types of transformations are possible, combined with excellent traversal and data acquisition which can be contributed to the power of strategic programming, all transformation scopes are easy to implement. Most noteworthy is the good support for global target transformations, which can largely be contributed to scoped dynamic rewrite rules, which provide a nice solution for certain types of global target problems. Furthermore, the availability of reverse transformations provides developers with yet another choice when developing transformations. We can conclude Stratego is a very powerful transformation tool suitable for implementing all discussed types of transformations proving its versatility and applicability in all kinds of different projects.

6.3 Evaluation result

We can draw a number of conclusions from this evaluation. If we look at the program transformation tools we can see the 1-to-1 scoped transformations have the best support, followed by global-to-local transformations. This is easy to explain by noting that these are conceptually the easiest transformations. Furthermore, most paradigms allow for an easy and intuitive approach by having a simple traversal with some support to rewrite a term to another term for 1-to-1 scoped transformations and simple data acquisition support to be able to implement global-to-local transformations. Combining this observation with the scopes of typical program transformations in Section 5.2, we see that indeed a lot of these transformations have a 1-to-1 or global-to-local scope, however many of them also need transformations with a different scope.

Another very interesting conclusion is that overall support for global target transformations is not very strong, while this type of transformation is needed by a number of typical program transformations. The reason for this is probably that it is not yet completely clear how to implement these in a clean way in different programming paradigms, making this is an area that needs further research. Another reason is that we think, after reading a lot of literature regarding program transformations, that up to now this type of transformation has received little attention.

Support for reverse transformations is also weak amongst the studied tools. This can largely be contributed to weak parser support. Traditional parser technology does not easily allow for mixing a meta-language with a custom output language, which forces many transformation tool developers to choose not to support concrete syntax at the expense of not supporting powerful reverse transformations. In [29] it is shown how the syntax definition formalism SDF can be used to fit any (meta-)programming language with concrete syntax for an arbitrary output language. Another likely reason for this weak support is that this type of transformations is simply overlooked by transformation system developers, just as the global target transformations.

In this section we have shown an interesting application of transformation mechanics in a large systematic evaluation of transformation tools. Again we want to make clear an evaluation like this is never complete and it is possible to introduce many more interesting features and aspects to use in this evaluation. However, we believe we have succeeded in showing the usefulness of program transformation aspects and raising some awareness of different types of program transformations and the need for tools to provide sufficient support for these.

7 Concluding Remarks

Contribution In this paper we have introduced a novel way of distinguishing different types of transformations using high-level transformation mechanics. This allows for several interesting applications that were not possible before. From a research perspective it provides a better general understanding of different types of transformations, providing a starting point for transformation design patterns and giving researchers a better insight in the strengths and weaknesses of different transformation tools.

From a practical perspective it can provide transformation system developers structure for their documentation. It now becomes possible to describe for which types of transformations a tool can be used for in a structured way and, maybe even more importantly, provide explanation, examples and design guidelines on how to implement them. The user of a transformation tool can use transformation mechanics in the design of a transformation and in selecting the tool that has the best support for the needed transformation. Also, because of more structured documentation, it will be easier to convert a high-level design into an implementation. Although this all sounds very obvious, at this moment it is still far from trivial.

Future Work Although the transformation mechanics can be used for both transformation implementation and transformation problems, the last application shows an interesting issue. Looking again at the constant propagation example, observe that the scope aspect can be influenced by the choice of pivot. If a constant variable is chosen to be the pivot this transformation has a local-to-globalscope, but if all variables in a block are chosen as pivot this transformation has a global-to-local scope. More research is needed to find out if different or more precise definitions can solve such ambiguities.

Another area for further research are the presented mechanics themselves. We provided an abstract view based on three key aspects of transformations. These key aspects were determined mainly based on our own experiences, but it is very well possible other general applicable or application dependent transformation aspects can be defined.

Acknowledgments Martin Bravenboer, Eelco Dolstra and Karina Olmos gave useful feedback on a previous version of this paper.

References

[1] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.

[2] J. M. Boyle. Abstract programming and program transformation—An approach to reusing programs. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press, 1989.

[3] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software.

In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, 1997.

[4] M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 237–251, Copenhagen, Denmark, July 2002. Springer-Verlag.

[5] S. Chiba. A metaobject protocol for C++. In *In Proceedings of the 10 th Conference on Object-Oriented Programming Systems, Languages and Programming*, pages 285–299, 1995.

[6] J. Cordy, T. Dean, A. Malton, and K. Schneider. Software engineering by source transformation - experience with txl. In *SCAM'01 - IEEE 1nd International Workshop on Source Code Analysis and Manipulation*, pages 168–178, Florence, Novenber 2001.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

[8] T. Dean, J. Cordy, A. Malton, and K. Schneider. Grammar programming in txl. In *SCAM'02 - IEEE 2nd International Workshop on Source Code Analysis and Manipulation*, Montreal, October 2002.

[9] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[12] G. Hedin and E. Magnusson. Jastadd - a java-based system for implementing front ends. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[13] S. E. Keller, J. A. Perkins, T. F. Payton, and S. P. Mardinly. Tree transformation techniques and experiences. In *ACM SIGPLAN '84 - Symposium on Compiler Construction*, pages 190–201, June 1984.

[14] D. E. Knuth. Semantics of context-free languages. In *Mathematical Systems Theory*, volume 2, pages 168–178, 1968.

[15] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In A. SIGPLAN, editor, *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*. ACM Press, 2003. To appear in ACM SIGPLAN Notices.

[16] R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming, 2002.

[17] R. Lämmel and J. Visser. Typed combinators for generic traversal, 2002.

[18] G. Lindén, H. Tirri, and A. I. Verkamo. Alchemist: A general purpose transformation generator, 1995.

[19] Microsoft. Active Server Pages (ASP). http://msdn.microsoft.com/asp.

[20] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[21] R. J. Rodger. Jostraca: a template engine for generative programming. *European Conference for Object-Oriented Programming*, 2002.

[22] S. Swierstra and P. Azero. Attribute grammars in the functional style. In *SI2000 Chapman-Hall*, 1998.

[23] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. Openjava: A class-based macro system for java. In *OORaSE*, pages 117–133, 1999.

[24] M. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[25] M. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions, 2001.

[26] J. van Wijngaarden. Code generation from a domain specific language. Master's thesis, Utrecht University, 2003. To Appear.

[27] E. Visser. Scoped dynamic rewrite rules. *Electronic Notes in Theoretical Computer Science*, 59(4), 2001.

[28] E. Visser. A survey of strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[29] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. 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.

[30] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

[31] J. Visser. Visitor combination and traversal control. In *OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 270–282, 2001.

[32] W3C. Xml path language (xpath) version 1.0. w3c recommendation 16 november 1999. http://www.w3.org/TR/xpath.

[33] W3C. Xsl transformations (xslt) version 1.0. w3c recommendation 16 november 1999. http://www.w3.org/TR/xslt.