ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

Journal of Symbolic Computation 40 (2005) 831–873

Journal of Symbolic Computation

www.elsevier.com/locate/jsc

A survey of strategies in rule-based program transformation systems

Eelco Visser*

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands

Received 3 February 2003; accepted 14 December 2004
Available online 2 April 2005

Abstract

Program transformation is the mechanical manipulation of a program in order to improve it relative to some cost function and is understood broadly as the domain of computation where programs are the data. The natural basic building blocks of the domain of program transformation are *transformation rules* expressing a 'one-step' transformation on a fragment of a program. The ultimate perspective of research in this area is a high-level, language parametric, rule-based program transformation system, which supports a wide range of transformations, admitting efficient implementations that scale to large programs. This situation has not yet been reached, as trade-offs between different goals need to be made. This survey gives an overview of issues in rule-based program transformation systems, focusing on the expressivity of rule-based program transformation systems and in particular on transformation *strategies* available in various approaches. The survey covers term rewriting, extensions of basic term rewriting, tree parsing strategies, systems with programmable strategies, traversal strategies, and context-sensitive rules.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Program transformation; Transformation rule; Transformation strategy; Program representation; Term rewriting; Pattern matching; Extensions of term rewriting; Strategy annotations; Tree parsing; Attribute grammars; Strategy combinators; Tree traversal; Congruence operators; Generic traversal strategies; Context-sensitive rules

* Tel.: +31 30 253 4592; fax: +31 30 251 3791.
E-mail address: visser@acm.org.
URL: http://www.cs.uu.nl/~visser.

0747-7171/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2004.12.011

1. Introduction

Program transformation is the mechanical manipulation of a program in order to improve it relative to some cost function C such that $C(P) > C(tr(P))$, i.e., the cost decreases as a result of applying the transformation. The cost of a program can be measured in different dimensions such as performance, memory usage, understandability, flexibility, maintainability, portability, correctness, and satisfaction of requirements. In general, transformations should preserve the semantics of the program according to some appropriate definition of semantics (Pettorossi and Proietti, 1996a; Paige, 1996; Cousot and Cousot, 2002). However, in some applications, such as program evolution, the goal of a transformation may be to deliberately *change* the semantics of the program. Furthermore, a strict interpretation of program transformations restricts the term to *rephrasings*, i.e., transformations of a program to another program in the same language. Here we also consider *translations* to programs in another language. Such translations can be seen as rephrasings in a language that is the union of the source and target languages. While transformations can be achieved by manual manipulation of programs, in general, the aim of program transformation is to increase programmer productivity by *automating* programming tasks, thus enabling programming at a higher level of abstraction, and increasing maintainability and reusability of programs.

Thus, program transformation is understood here broadly as the domain of computation where programs are the data. In practice, the area is divided into many different schools corresponding to application areas and implementation techniques. Many transformation systems are built for a particular object language, a particular type of transformation, and for use in a particular environment. The implementation uses specific data structures and involves complex algorithms in order to achieve maximal performance, e.g., for use in optimizing compilers. The resulting ad hoc monolithic transformation systems are difficult to understand, maintain, and reuse.

The aim of a broad consideration of the field is the reuse of results from subfields to arrive at a unified high-level approach to the implementation of transformation systems. The ultimate goal of this endeavor is a component-based approach to program transformation in which basic transformation components can be reused in many different compositions. The natural 'basic components' of the domain of program transformation are *transformation rules* expressing a 'one-step' transformation on a fragment of a program. *Rule-based* program transformation systems support formulation of basic transformation rules and arrange their automatic application. Thus, the ultimate perspective of research in this area is a high-level, language parametric, rule-based program transformation system, which supports a wide range of transformations, admitting efficient implementations that scale to large programs.

This goal has not yet been reached, as trade-offs between different goals need to be made. The suitability of a rule-based transformation system for the implementation of a certain type of transformation depends on the expressivity in the formulation of rules, on the strategies available for their control, and on the quality of their implementation. A highly generic system may allow concise specification of many different transformations, but not with the speed of a hand-written optimization component. On the other hand, a dedicated tool with a restricted type of rule may be able to generate

highly optimized transformers, but may not be applicable to a slightly different type of transformation.

A special concern in rule-based systems is the definition of *strategies* for the application of rules. The combination of rules into complete transformations requires control over the application of rules. While rule-based systems traditionally apply rules *automatically* according to a *standard control strategy*, it turns out that program transformation usually requires more careful control. Thus, rule-based transformation systems tend to adopt mechanisms for controlling the application of rules.

To summarize, research in the area of rule-based program transformation systems is concerned with:

- Formulation of rule-based solutions to a wide range of transformation problems.
- Concise and reusable specification of rules and strategies.
- Generation of efficient implementations for rule-based specifications (e.g., by adopting implementation techniques from transformation systems for more specific domains).

This survey gives an overview of issues regarding rule-based program transformation systems, focusing on the second item above, i.e., the expressivity of rule-based program transformation systems, and in particular on transformation *strategies* available in various approaches.

To set the scene the next section describes the wide range of applications of program transformation. Section 3 then describes *term rewriting* as the basis for rule-based program transformation. At some appropriate level of abstraction all program transformations can be modeled as the consecutive application of rewrites, even though this model may not always be visible in the actual implementation where rules and strategies are blended into a monolithic implementation for efficiency and other reasons. The basic approach to term rewriting with standard rewriting strategies such as innermost and outermost has a number of limitations for application in transformation systems. The remaining sections discuss extensions to the basic approach.

Section 4 discusses various ways of expressing properties of the syntax and semantics of the programming language in rewrite rules. Examples include the extension of term rewriting with *concrete syntax*, *equational matching*, *bound object variables*, *default rules*, and *strategy annotations*. Section 5 discusses approaches based on *tree parsing* in which *tree grammar rules* define actions to be performed on tree nodes. A tree traversal schedule is computed based on the dependencies between rules.

Section 6 considers the development of interactive systems for the assistance of transformational programming, in which the need arose to automate reoccurring sequences of transformations. This gave rise to systems with programmable transformation strategies. A particular concern in the specification of strategies is the traversal of program structures. Section 7 gives an overview of the various solutions developed in this area.

Another shortcoming of rewrite rules is their context-free nature. That is, rules only have access to the information in the term they apply to. Section 8 presents solutions to this problem, which include information propagating strategies and the dynamic generation of rewrite rules.

Finally, there are many other issues that play a role in program transformation systems and other approaches that are beyond the scope of this article.

2. Applications of program transformation

Program transformations can be classified according to various criteria such as amount of automation, improvement achieved, and subject language (Feather, 1987; Partsch, 1990; Smaragdakis and Batory, 2000). When considered along the lines of research communities, roughly two main schools can be distinguished, i.e., those concerned with the *development* of new programs and those concerned with the *evolution* of existing programs.

2.1. Program development

Program development is concerned with the transformation from requirements via specification to implementation. Usually the initial parts of the process involve intervention from a programmer or even a software architect, while later stages in the process are completely automated. In the course of the last fifty years the boundary of automation has shifted considerably; *for*mula *tran*slation was considered an innovative automation in the 1950s, while arithmetic expressions are the assembly language (lowest level of abstraction) for today's programmers.

Transformational programming (Partsch, 1986; Feather, 1987) is a methodology for formal development of implementations from specifications and is on the boundary of automation, i.e., *formal* means that it can be mechanized (but not necessarily automated), *development* entails traceability. In the course of development, design information is traded for increased efficiency. Feather (Feather, 1987) distinguishes *meta-programming*, i.e., the interactive transformation of a specification into an implementation; *extended compilation*, i.e., completely automatic compilation for a language with high-level constructs with advice from the programmer to the compiler about a specific program; and *program synthesis* or *refinement* (Smith, 1990) make transformation as automatic as possible without limiting the specification language in any way.

Compilers provide completely automatic transformation from a high-level language to a low-level language (Aho et al., 1986; Appel, 1998; Muchnick, 1997). This translation is usually achieved in several phases. Typically, a high-level language is first translated into a target machine independent intermediate representation. Instruction selection then translates the intermediate representation into machine instructions. In *transformation-based compilers* such as GHC (Peyton Jones and Santos, 1998) a large part of the compilation process is implemented as the application of small transformation steps. GHC even allows the programmer to specify additional rules for application in the compiler (Peyton et al., 2001). *Application generators* (Smaragdakis and Batory, 2000) are compilers for domain-specific languages. Examples are parser and pretty-printer generation from context-free grammars (Aho et al., 1986; van den Brand and Visser, 1996). A *program optimization* (Appel, 1998; Muchnick, 1997) is a transformation that improves the run-time and/or space performance of a program. Example optimizations are fusion, inlining, constant propagation, constant folding, common-subexpression elimination, dead code elimination, and partial evaluation (Jones et al., 1993).

2.2. Program evolution

Program evolution is concerned with the understanding and maintenance of existing *legacy* programs.

Reverse engineering (Chikofski and Cross, 1990; van den Brand et al., 1997) is the inverse of compilation, i.e. the goal is to extract from a low-level program a high-level program or specification, or at least some higher-level aspects. Reverse engineering raises the level of abstraction and is the dual of synthesis. Examples of reverse engineering are decompilation in which an object program is translated into a high-level program, architecture extraction in which the design of a program is derived, documentation generation, and software visualization in which some aspect of a program is depicted in an abstract way.

In *software renovation* the extensional behavior of a program is changed in order to repair an error or to bring it up to date with respect to changed requirements. Examples are repairing a Y2K bug, or converting a program to deal with the Euro. *Refactoring* (Fowler, 1999) is renovation aimed at improving the design of a program by restructuring it such that it becomes easier to understand while preserving its functionality. *Obfuscation* (Collberg et al., 1998) is a transformation that makes a program *harder* to understand by renaming variables, inserting dead code, etc. Obfuscation is done to hide the business rules embedded in software by making it harder to reverse engineer the program.

In *migration* a program is transformed to another language at the same level of abstraction. This can be a translation between dialects, for example, transforming a Fortran77 program to an equivalent Fortran90 program or a translation from one language to another, e.g., porting a Pascal program to C.

3. Term rewriting

A complex program transformation is achieved through a number of consecutive modifications of a program. At least at the level of design, it is useful to distinguish transformation rules from transformation strategies. A *rule* defines a basic step in the transformation of a program. A *strategy* is a plan for achieving a complex transformation using a set of rules.

This section first examines the conceptual notion of transformation rules and strategies, then considers the issue of representation of programs, and finally describes the implementation of transformation by means of term rewriting and its limitations.

3.1. Transformation rules and strategies

Rules are based on the semantics of the language. A rule generally preserves the semantics of the program. That is, before and after the application of a rule the program has the same meaning. Usually the observable behavior of the program is preserved, but some other aspect is changed. Optimizations, for example, try to decrease the time or space resource usage of a program. Applying constant propagation can decrease the need for registers, for instance. Extracting a function during refactoring can improve the readability of the program.

```
rules
  InlineF :
    [[ let f(xs) = e in e'[f(es)] ]] -> [[ let f(xs) = e in e'[e[xs := es]] ]]
  InlineV :
    [[ let x = e in e'[x] ]] -> [[ let x = e in e'[e] ]]
  Dead :
    [[ let x = e in e' ]] -> [[ e' ]] where x ∉ e'
  Extract(f,xs) :
    [[ e ]] -> [[ let f(xs) = e in f(xs) ]]
  Hoist :
    [[ let x = e1 in let f(xs) = e2 in e3 ]] ->
    [[ let f(xs) = e2 in let x = e1 in e3 ]]
    where x ∉ free-vars(e2)
```

Fig. 1. Some example transformation rules.

A rule involves recognizing a program fragment to transform and constructing a new program fragment to replace the old one. Recognition involves matching the structure of the program and possibly verifying some semantic conditions. The replacement in a rule can consist of a simple term pattern, a function that constructs a new tree or graph, or a semantic action with arbitrary side-effects.

Consider the transformation rules in Fig. 1. The `Inline` rules define inlining of function and variable definitions. The `Dead` rule eliminates an unused variable definition. The `Extract` rule abstracts an expression into a function. The `Hoist` rule defines lifting a function definition out of a variable definition if the variable is not used in the function. Using this set of rules different transformations can be achieved. For example, a constant propagation strategy in an optimizer could use the `InlineV` and `Dead` rules to eliminate constant variable definitions:

```
let x = 3 in x + y -> let x = 3 in 3 + y -> 3 + y
```

On the other hand, a function extraction strategy in a refactoring browser could use the `Extract` and `Hoist` rules to abstract addition with `y` into a new function and lift it to top-level.

```
let x = 3 in x + y
-> let x = 3 in let addy(z) = z + y in addy(x)
-> let addy(z) = z + y in let x = 3 in addy(x)
```

A set of transformation rules for a programming language induces a rewrite relation on programs (Dershowitz and Jouannaud, 1990). If the relation is confluent and terminating, there is a unique normal form for every program. In that case it is a matter of applying the rules in the most efficient way to reach the normal form. However, in program transformation this is usually not the case. As illustrated in Fig. 2, a set of transformation rules can give rise to infinite branches (e.g., by inlining a recursive function), inverses in which a transformation is undone (e.g., by distribution or commutativity rules), and non-confluence in which a program can be transformed into two different programs.

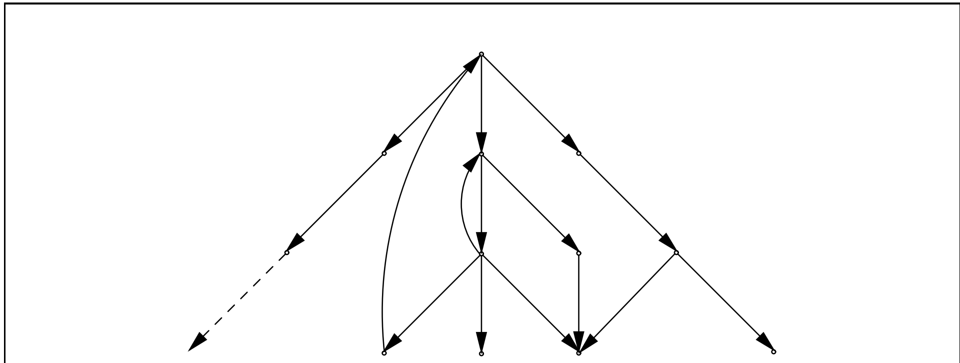

Fig. 2. Phenomena in composition of transformation rules: infinite branches, inverses, confluence, non-confluence.

Depending on the goal of a transformation task, a path should be chosen in the rewrite relation. For a specific program it is always possible to find the shortest path to the optimal solution for a specific transformation task. However, for most transformation tasks the process of finding a path needs to be automated and optimal solutions might only be approximated. In this light, a strategy is *an algorithm for choosing a path in the rewrite relation*. Given one set of rules, there can be many strategies, each achieving a different goal. On the other hand, a general strategy can be applicable to many different sets of rules.

3.2. Program representation

Before examining in more detail how rules and strategies can be defined, we need to consider how the programs they manipulate are represented. Design decisions made at the level of representation influence the design decisions that can be made in the definition of rules and strategies.

Although some systems work directly on text, in general a textual representation is not adequate for performing complex transformations. Therefore, a structured representation is used by most systems, and only such systems are studied in this survey. Since programs are written as text by programmers, parsers are needed to convert from text to structure and unparsers are needed to convert structure to text. Since such tools are well covered elsewhere (Aho et al., 1986), they are not examined in this survey.

3.2.1. Parse trees or abstract syntax trees

A parse tree is a direct representation of the derivation of a string (the program text) according to the rules of a grammar. Parse trees contain syntactic information such as layout (white space and comments), and parentheses and extra nodes introduced by disambiguating grammar transformations. Since this information is often irrelevant for transformation, parse trees are usually transformed into abstract syntax trees that do not contain such information. However, for some applications (such as software renovation and refactoring) it is necessary to restore as much as possible the original layout of the program after transformation. This requires that layout is stored in the tree and preserved

throughout transformation. A similar issue is the storage of source locations (line and column numbers) in order to provide useful error messages. The preservation of layout and position information is especially problematic; it is not clear in a generic manner where to insert comments in a transformed fragment of a program. Possible solutions to this problem include origin tracking (van Deursen et al., 1993) in which a subtree in the transformed program is related to the subtree it 'originates' from in the original tree; parse-tree annotations (Kort and Lämmel, 2003) that carry their own methods for propagation; and comparing the transformed program to the original to infer where comments can be inserted.

For other applications, e.g., certain optimizations and compilation, it is necessary to carry type information in the tree. This requires the extension of the tree format to store type information and to preserve consistency of types throughout transformation.

3.2.2. Trees or graphs

Program structure can be represented by means of trees, directed acyclic graphs (DAGs), or full fledged graphs with cycles.

Using pure trees is costly because copying of a tree (e.g., by using a variable twice when constructing a new tree) requires creating a complete copy. Therefore, most systems use DAGs. When copying a tree, only a pointer to the tree gets copied; thus subtrees are shared by multiple contexts. The advantage of sharing is reduced memory usage. In the ATerm library (van den Brand et al., 2000a) this approach is taken to the extreme by only constructing one instance for each subtree that is constructed, thus achieving maximal sharing and minimal memory usage. Furthermore, testing the equality of two terms becomes an O(1) operation.

Sharing saves memory, makes copying cheap, and, in the case of maximal sharing, testing for equality is cheap as well. However, the downside of sharing is that performing a transformation of a tree requires rebuilding the context in which the transformed tree is used. It would be more attractive to overwrite the root node of the subtree that is changed with the new tree, thus updating all contexts in which the old tree was used. However, this is not valid in general. Two occurrences of a shared tree that are syntactically the same can have a completely different meaning depending on their context. Even if they have the same meaning, it is not always desirable to change both occurrences.

The same problem of occurrence arises when associating information with nodes. When sharing is based on syntactic equivalence alone, annotations become associated with all occurrences of the tree. Consider the examples of position information in parse trees and type annotations in abstract syntax trees to conclude that this is usually not desirable. On the other hand, if annotation of a tree node results in a new tree, then equivalence becomes equivalence with annotations, and equivalence modulo annotations is no longer a constant operation.

Finally, full fledged graphs can be useful for representing backlinks in the tree to represent, for example, loops in a control-flow graph (Appel, 1998; Lacey and de Moor, 2001; Muchnick, 1997), or links to declarations (Czarnecki and Eisenecker, 2000). Updateable graphs make it easy to attach new information to nodes, for example results of analysis. The problem of destructive update versus copying while doing transformation is even more problematic in graphs. Since a subgraph can have links to the entire graph, it may be required to reconstruct the entire graph after a transformation if it is necessary

to keep the original graph as well. For very specific purposes such as lazy evaluation of functional programs, it is possible to make such graph updates transparent.

3.3. Term rewriting

Term rewriting is a good starting point for the study of program transformation systems. Term rewriting is a simple formalism modeling modification of trees (terms) through a sequence of rewrite rule applications. Thus, providing a general model for program transformation. Any *specific* transformation can be modeled as a sequence of rewrites on the program tree. This does not necessarily mean that such a sequence can always be seen as the normalizing application of a set of rewrite rules according to a standard strategy. That is, term rewriting interpreted as *exhaustive application of a set of rules* is not an adequate technique for all applications of program transformation. The rest of this section describes term rewriting and its limitations for use in program transformation. The description is limited to the basics of term rewriting; introductions to the vast literature on rewrite systems include Dershowitz and Jouannaud (1990), Klop (1992), Baader and Nipkow (1998) and Terese (2003).

A *term rewriting system* is a collection of *rewrite rules* defining one-step transformations of *terms*. Terms are symbolic representations for the structure to be transformed. We first consider basic term rewriting with first-order terms representing trees or DAGs. More complex term structures will be discussed in the next section.

3.3.1. Terms

An *algebraic signature* defines a family of sorted first-order terms through a set of *constructor declarations* as follows: If $C : S_1 * \cdots * S_n \to S_0$ is a constructor declaration in the signature and t_1 is a term of sort $S_1, \ldots, t_n$ a term of sort S_n, then $C(t_1, \ldots, t_n)$ is a term of sort S_0. Note that $C : S$ is shorthand for $C :\to S$.

First-order terms can be used to describe the abstract syntax trees of programs. There is a one-to-one correspondence between first-order terms and trees with constructors as node labels and an ordered set of directed edges to the trees corresponding to the subterms. Directed acyclic graphs can be used to efficiently represent sharing in terms. Fig. 3 illustrates this with a signature for the language of propositional formulae. For instance, the formula $p \wedge \neg q$ is represented by the term `And(Atom("p"), Not(Atom("q")))`. Note that the sort `String` is used to represent the set of all character strings. Another syntactic extension of first-order terms that are indispensable in program transformation are lists of the form `[`t_1`,...,`t_n`]` which abbreviate terms of the form `[`t_1`|[`t_2`|...[`t_n`|[]]]]`, i.e. terms over the signature

```
[]    : List(a)
[_|_] : a * List(a) -> List(a)
```

3.3.2. Rewrite rules

A *rewrite rule* is a pair of term patterns written as p_1 `->` p_2. A *term pattern* is a term with *variables*. A *labeled* rewrite rule is a named rule of the form L : p_1 `->` p_2. A rule defines a transformation of an expression of the form p_1 to an expression of the form p_2. For example, the rule

```
signature
  sorts Prop
  constructors
    False : Prop
    True  : Prop
    Atom  : String -> Prop
    Not   : Prop -> Prop
    And   : Prop * Prop -> Prop
    Or    : Prop * Prop -> Prop
rules
  DAOL  : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR  : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL  : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR  : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN    : Not(Not(x))      -> x
  DMA   : Not(And(x, y))   -> Or(Not(x), Not(y))
  DMO   : Not(Or(x, y))    -> And(Not(x), Not(y))
```

Fig. 3. Signature and rewrite rules for propositional formulae.

```
AA : And(And(x, y), z) -> And(x, And(y, z))
```

associates conjunction to the right. Rewrite rules can be used to express basic transformation rules and can be considered as operationalizations of the algebraic laws of the language. For example, the rewrite rules in Fig. 3 express basic laws of propositional logic, i.e., the distribution rules, the rule of double negation, and the De Morgan rules.

3.3.3. Reduction

A rule L : p_1 `->` p_2 *reduces* a term t to t', if t matches p_1 with a substitution σ, i.e., $t = \sigma(p_1)$, and $t' = \sigma(p_2)$. We say that t is the *redex* (reducible expression), and t' is the *reduct*. Thus, with rule `AA` we have the reduction

```
    And(And(Var("a"), False), Var("b"))
 -> And(Var("a"), And(False, Var("b")))
```

since the substitution [`x` := `Var("a")`, `y` := `False`, `z` := `Var("b")`] defines a match for the left-hand side `And(And(x, y), z)` of the rule, and instantiates the right-hand side `And(x, And(y, z))` to the reduct.

A set of rewrite rules R induces a *one-step rewrite relation* on terms. If t reduces to t' with one of the rules in R then we have $t \rightarrow_R t'$. In this relation reductions take place at the root of terms. The relation can be extended to the relation $\Rightarrow_R$ which relates two terms with a reduction under the root. The relation is formally defined as follows:

$$\frac{t_1 \rightarrow_R t_2}{t_1 \Rightarrow_R t_2} \qquad \frac{t_i \Rightarrow_R t_i'(1 \leq i \leq n)}{c(t_1, ..., t_i, ..., t_n) \Rightarrow_R c(t_1, ..., t_i', ..., t_n)}.$$

For example, with rule `A : P(Z, x) -> x` the term `P(P(Z,S(Z)), S(Z))` reduces to `P(S(Z), S(Z))` by reducing the first argument of the outermost `P`.

A term t *rewrites* to a term t' with respect to a set of rewrite rules R if there is a finite sequence of terms $t = t_1, \ldots, t_n = t'$ such that each t_i reduces (under the root) to t_{i+1}.

This is formalized by the rewrite relation $\Rightarrow^*_R$, defined by the following rules:

$$\frac{}{t \Rightarrow^*_R t} \qquad \frac{t_1 \rightarrow_R t_2}{t_1 \Rightarrow^*_R t_2} \qquad \frac{t_1 \Rightarrow^*_R t_2 \quad t_2 \Rightarrow^*_R t_3}{t_1 \Rightarrow^*_R t_3}$$

$$\frac{t_i \Rightarrow^*_R t'_i (1 \leq i \leq n)}{c(t_1, \ldots, t_i, \ldots, t_n) \Rightarrow^*_R c(t_1, \ldots, t'_i, \ldots, t_n)}.$$

That is, the reflexive, transitive and congruent closure of $\rightarrow_R$.

A term t is in *normal form* with respect to a set of rewrite rules R, if there is no term t' not equal to t such that $t \Rightarrow^*_R t'$. If the rules in R are unconditional, this is the case if there is no subterm of t that matches with one of the left-hand sides of a rule in R.

3.3.4. Rewriting strategies

The reduction relation induced by a set of rewrite rules is a tool for mechanizing the transformation of programs. Given a set of rewrite rules, correct according to some criterion, a program can be transformed by applying the rules in the order needed for the specific transformation. Thus, rewriting can be used to *model* any specific transformation. However, this does not provide us with a procedure for performing such transformations automatically; the reduction relation does not impose any order on the application of rules.

A *rewriting strategy* is an algorithm for applying rules to achieve a certain goal. Typically the goal is to *normalize* a term with respect to a set of rules, that is, exhaustively apply rules to the term until it is in normal form. One popular strategy for normalization is *innermost* normalization, as defined by the relation $\Rightarrow^{im}_R$:

$$\frac{t_1 \Rightarrow^{im}_R t'_1 \;\ldots\; t_n \Rightarrow^{im}_R t'_n \quad c(t'_1, \ldots, t'_n) \Rightarrow^{red}_R t}{c(t_1, \ldots, t_n) \Rightarrow^{im}_R t}$$

$$\frac{t_1 \rightarrow_R t_2 \quad t_2 \Rightarrow^{im}_R t_3}{t_1 \Rightarrow^{red}_R t_3} \qquad \frac{\neg\exists t_2 : t_1 \rightarrow_R t_2}{t_1 \Rightarrow^{red}_R t_1}.$$

This strategy states that before applying a rule to a term, first all its subterms are normalized. The *outermost* strategy, in contrast, first reduces redices closest to the root, as defined by the relation $\Rightarrow^{om}_R$:

$$\frac{t_1 \Rightarrow^{rom}_R t_2}{t_1 \Rightarrow^{om}_R t_2} \qquad \frac{t_1 \Rightarrow_R t_2^{om} \quad t_2 \Rightarrow^{om}_R t_3}{t_1 \Rightarrow^{om}_R t_3} \qquad \frac{t_1 \rightarrow_R t_2}{t_1 \Rightarrow^{rom}_R t_2}$$

$$\frac{\neg\exists t' : c(t_1, \ldots, t_i, \ldots t_n) \rightarrow_R t' \quad t_i \Rightarrow^{rom}_R t'_i (1 \leq i \leq n)}{c(t_1, \ldots, t_i, \ldots, t_n) \Rightarrow^{rom}_R c(t_1, \ldots, t'_i, \ldots, t_n)}.$$

This strategy is the transitive closure of the relation $\Rightarrow^{rom}_R$, which defines the reduction of a single outermost redex.

Normalization of terms with respect to a set of rewrite rules is applicable in areas such as algebraic simplification of expressions, and is provided by many rewriting engines, including OBJ (Goguen et al., 2000), ASF+SDF (van Deursen et al., 1996), ELAN (Borovanský et al., 1996), Maude (Clavel et al., 2002), Stratego (Visser, 2004) and

many others. An overview of rewriting-based languages is presented in Heering and Klint (2003). Most of these systems also support extensions of the basic rewriting paradigm.

3.4. Limitations of term rewriting

The advantage of term rewriting is that all that is needed for the implementation of a transformation is the specification of a set of rewrite rules. The rewriting engine takes care of traversing the program tree in order to find reducible expressions. In other words, term rewriting *separates rules and strategy*. Due to this property the size of the specification corresponds to the size of the problem to be solved, and is independent of the complexity of the language, i.e., the size of the signature.

However, the complete normalization approach of rewriting turns out not to be adequate for program transformation, because rewrite systems for programming languages will often be non-terminating and/or non-confluent. In general, it is not desirable to apply all rules at the same time or to apply all rules under all circumstances. As an example, consider again the set of rewrite rules in Fig. 3. This rewrite system is non-terminating because rules `DAOL` and `DAOR` enable rules `DOAL` and `DOAR`, and vice versa. If we want to define a transformation to normalize formulae to disjunctive normal form we could discard rules `DOAL` and `DOAR`. However, if in another part of the transformation a conjunctive normal form is required we need a different rewrite system. It is not possible to combine these rules in one rewrite system. Another example is the following perfectly valid rule

```
Unroll : While(e1, e2) -> If(e1, Seq(e2, While(e1, e2)))
```

defining the unrolling of a loop in an imperative language. Applying such a rule exhaustively directly leads to non-termination. It is not even possible to create a terminating system by leaving out other rules.

Thus, the basic approach of normalizing a program tree with respect to a set of transformation rules is not sufficient since no control over the application of rules is provided. To provide users with more control, various solutions have been adopted in transformation systems, ranging from alternative automatic strategies to completely user-definable strategies. We can distinguish the following approaches:

- Fixed application order. The engine applies rules exhaustively according to a built-in strategy. Examples are innermost and outermost reduction.
- Automatic dependency analysis. The engine determines a strategy based on an analysis of the rules. Examples are strictness and laziness analysis.
- Goal driven. The engine finds out how to apply rules to achieve a user-defined goal.
- Strategy menu. A strategy can be selected from a small set. For example, choose between innermost and outermost reduction or annotate constructors with laziness information.
- Programmable. The strategy to apply rules can be programmed in a strategy language.

In addition, there are a number of other shortcomings of basic term rewriting:

- Term syntax is not easy to read and write when terms (program fragments) become large. This may seem a minor issue, but it is relevant in program transformation.

- Basic term pattern matching is not very expressive and cannot cope with properties of constructors such as associativity and commutativity.
- Object variables are treated as normal terms and require careful handling to avoid name capture.
- Transformation rules often need side conditions to test the applicability of the transformation.
- Generic rule-based solutions do not always provide the required performance needed for application in, say, production compilers.
- Rewrite rules are context-free, i.e., can only access information through pattern matching the term to which the rule is applied. Often context information is needed for transformations.

For these reasons many extensions and variations on the basic paradigm of rewriting have been developed for the application in program transformation systems. Also program transformation systems are built using non-rewrite systems, in which the same issues play a role. The rest of this survey examines the solutions for these problems employed in a variety of transformation systems. Although the emphasis is on control issues, the other problems mentioned above are discussed as well since they are recurring problems in transformation systems and solutions may interfere with solutions for the control problem.

4. Extensions of term rewriting

In this section we consider several extensions to basic term rewriting that make the formalism more expressive.

4.1. Concrete syntax

Related to the internal representation of programs is the representation of program fragments in the *specification* of transformation rules. While abstract syntax provides a good model for program transformation, the direct manipulation of abstract syntax trees may not be appropriate. Abstract syntax trees are represented using the data structuring facilities of the transformation language: records (structs) in imperative languages (C), objects in object-oriented languages (C++, Java), algebraic data types in functional languages (ML, Haskell), and terms in term rewriting systems.

Such representations allow the full capabilities of the transformation language to be applied in the implementation of transformations. In particular, when working with high-level languages that support symbolic manipulation by means of pattern matching (e.g., ML, Haskell) it is easy to compose and decompose abstract syntax trees. For transformation systems such as compilers, programming with abstract syntax is adequate; only small fragments, i.e., a few constructors per pattern, are manipulated at a time. Often, object programs are reduced to a core language that only contains the essential constructs. The abstract syntax can then be used as an intermediate language, such that multiple languages can be expressed in it, and transformations can be reused for several source languages.

However, there are many applications of program transformation in which the use of abstract syntax is not satisfactory since the conceptual distance between the concrete programs that we understand and the data structure access operations used for composition

and decomposition of abstract syntax trees is too large. This is evident in the case of record manipulation in C, where the construction and deconstruction of patterns of more than a couple of constructors becomes unreadable. But even in languages that support pattern matching on algebraic data types, the construction of large code fragments in a program generator can become painful.

Transformation languages supporting *concrete object syntax* let the programmer define transformations using the concrete syntax of the object language, while internally using abstract syntax trees. For example, in Stratego the loop unrolling rule from the previous section can be written as

```
Unroll :
  |[ while e1 do e2 ]| -> |[ if e1 then (e2; while e1 do e2) ]|
```

where the `|[...]|` delimiters are used to embed fragments of the object language as terms in rewrite rules. This approach was developed in the algebraic specification community. Using the correspondence between a context-free grammar and an algebraic signature (Hatcher and Rus, 1976; Goguen et al., 1977; Futatsugi et al., 1985), a constructor can be declared as *mixfix* operator, e.g., `if e1 then e2 else e3` instead of `If(e1,e2,e3)`. Although available in systems such as OBJ, ELAN, and Maude, the approach is taken to its extreme in ASF+SDF (Heering et al., 1989; van Deursen et al., 1996), where an actual syntax definition of the object language is used to describe terms (rather than just mixfix operators). The approach is further generalized in Visser (2002), Fischer and Visser (2004) and Bravenboer and Visser (2004), where a general scheme for extending a meta-language with concrete object syntax is outlined.

Another line of work is that of meta-programming languages such as MetaML (Taha and Sheard, 2000) and Template Haskell (Sheard and Peyton Jones, 2002) where fragments of a program can be transformed or generated *in the language itself*. These fragments can be written in concrete syntax rather than abstract syntax, but a fall back to abstract syntax is available when necessary.

4.2. Extensions of pattern matching

When using a tree or term representation *term pattern matching* can be used. First-order term patterns are used to decompose terms by simultaneously recognizing a structure and binding variables to subterms, which would otherwise be expressed by nested conditional expressions that test tags and select subterms. However, first-order patterns are not treated as first-class citizens and their use poses limitations on modularity and reuse: no abstraction over patterns is provided because they may occur only in the left-hand side of a rewrite rule, the arms of a case, or the heads of clauses; pattern matching is at odds with abstract data types because it exposes the data representation; a first-order pattern can only span a fixed distance from the root of the pattern to its leaves, which makes it necessary to define recursive traversals of a data structure separately from the pattern to get all needed information.

For these reasons, enhancements of the basic pattern matching features have been implemented or considered for several languages. For example, *list matching* in ASF+SDF (van Deursen et al., 1996) is used to divide a list into multiple sublists possibly separated by element patterns. *Associative–commutative (AC) matching* in Maude

(Clavel et al., 2002) and ELAN (Borovanský et al., 1996) supports the treatment of lists as multisets. Languages for XML transformation such as CDuce (Benzaken et al., 2003) provide *recursive patterns* and *regular expression patterns* to match complex subdocuments. *Higher-order unification* in λProlog (Nadathur and Miller, 1988; Pfenning and Elliot, 1988) allows higher-order matching of subterms in an arbitrary context (Felty, 1992; Heering, 1992), which in turn allows matching of subterms at arbitrarily deep levels using higher-order variables without explicit traversal of the structure involved. The MAG transformation system (de Moor and Sittampalam, 2001) supports a restricted form of higher-order matching to specify generic fusion rules. *Views* for Haskell, as proposed in (Wadler, 1987), provide a way to view a data structure using different patterns than are used to represent them. *Overlays* in Stratego (Visser, 1999) are pseudo-constructors that abstract from an underlying representation using actual constructors. The *contextual* and *recursive patterns* of Stratego (Visser, 1999) are in fact *strategies* for tree traversal. Thus, pattern matching and strategic control overlap.

4.3. Object variables

A particular problem of program transformation is the handling of variables and variable bindings. In the common approach, variables and variable bindings in an abstract syntax tree are treated just like any other construct and the transformation system has no special knowledge of them. This requires the implementation of operations to rename bound variables, substitution, etc. Transformations need to be aware of variables by means of extra conditions to avoid problems such as free variable capture during substitution and lifting variable occurrences out of bindings.

Transparent handling of variable bindings is desirable. In the use of De Bruijn terms (de Bruijn, 1980), bound variable *names* are replaced with indices pointing to the binding construct. This has the nice property that equivalence modulo renaming becomes syntactic equivalence. However, the scheme is hard to understand when reading program fragments. Furthermore, when transforming De Bruijn terms, the indices need to be recomputed. Higher-order abstract syntax (HOAS) (de Moor and Sittampalam, 2001; Huet and Lang, 1978; Pfenning and Elliot, 1988) gives a solution to such problems by encoding variable bindings as lambda abstractions. In addition to dealing with the problem of variable capture, HOAS provides higher-order matching which synthesizes new functions for higher-order variables. One of the problems of higher-order matching is that there can be many matches for a pattern, requiring a mechanism for choosing between them. FreshML (Pitts and Gabbay, 2000) provides a weaker mechanism for dealing with variable bindings that transparently refreshes variable names, thus solving the capture problem. Substitution for variables has to be dealt with explicitly. Both HOAS and FreshML require some amount of encoding for the syntactic structure to fit the lambda abstraction binding scheme. This can become rather far removed from the structure described by the grammar for more complex binding schemes. Furthermore, implicit variable binding may be in conflict with the ease of performing transformations, for instance, the possibility of performing traversals over syntax trees.

Experience with variable renaming in GHC, the transformation-based Glasgow Haskell Compiler (Peyton Jones and Marlow, 2002), shows that transparent treatment of variable

bindings would help only in a few places in the compiler. A problem encountered there was to minimize the amount of variable renaming done during transformation. Rather than using a global fresh variable store, fresh names are generated with respect to the *in scope* variables only.

In applications such as refactoring and renovation it is required that the transformed code is as close as possible to the original code. Approaches to dealing with object variables by renaming, are in conflict with this requirement.

A problem that is not addressed by the approaches discussed above is associating declaration information, e.g., type declarations, with usage. This usually requires maintaining a symbol table during transformation, or distributing the information over the tree, annotating usage occurrences of a symbol with the information in the declarations. Either way, the information needs to be kept consistent during transformations.

4.4. Default rules

A term rewrite system consists of a *set* of rewrite rules. This means that there is no inherent ordering of rules. Hence, an implementation can apply rules in any order. Although an implementation needs to choose *an* order, the programmer is not supposed to use this ordering since that violates the declarative nature of the rules. For example, the following rules

```
Mem1 : Member(x, [])        -> False
Mem2 : Member(x, [x | xs]) -> True
Mem3 : Member(x, [y | xs]) -> Member(x, xs)
```

rewrite applications of the `Member` function by first testing with the nonlinear rule `Mem2` whether the first element of the list is equal to the element looked for. The `Mem3` rule rewrites the application to a search in the tail of the list *assuming* that the match with the previous rule has failed. However, rule `Mem3` is not valid by itself. A rewrite engine may change the order of applying the rules, leading to unanticipated results.

A solution to the problem of implicitly ordering rewrite rules, which has been adopted in some systems such as ASF+SDF (van Deursen et al., 1996), is the declaration of *default* rules. A default rule is tried in a match only after all other (non-default) rules have been tried *at the current term*. Thus, the rewrite rules above can be ordered by declaring rule `Mem3` as a default rule:

```
Mem3 : Member(x, [y | xs]) -> Member(x, xs) (default)
```

Note that this declaration splits the set of rules into *two* sets. Default rules and non-default rules. *Priority rewriting* (Baeten et al., 1989) is a generalization of rewriting with default rules in which a partial order on rules is imposed.

4.5. Functional programming with rewrite rules

A common solution to the problem of control over the application of rules is the adoption of a functional programming style of rewriting. This is not so much an extension as a style of implementing transformation systems with rewrite rules. The method works by introducing additional constructors that achieve normalization under a restricted set of

```
signature
  constructors
    dnf : Prop -> Prop
    and : Prop * Prop -> Prop
    not : Prop -> Prop
rules
  DNF1  : dnf(True)       -> True
  DNF2  : dnf(False)      -> False
  DNF3  : dnf(Atom(x))    -> Atom(x)
  DNF4  : dnf(Not(x))     -> not(dnf(x))
  DNF5  : dnf(And(x,y))   -> and(dnf(x),dnf(y))
  DNF6  : dnf(Or(x,y))    -> Or(dnf(x),dnf(y))

  AND1  : and(Or(x,y),z) -> Or(and(x,z),and(y,z))
  AND2  : and(z,Or(x,y)) -> Or(and(z,x),and(z,y))
  AND3  : and(x,y)       -> And(x,y) (default)

  NOT1  : not(Not(x))    -> x
  NOT2  : not(And(x,y))  -> Or(not(x),not(y))
  NOT3  : not(Or(x,y))   -> and(not(x),not(y))
  NOT4  : not(x)         -> Not(x) (default)
```

Fig. 4. Functionalized rewrite system for disjunctive normal form.

rules. Such constructors are called *functions* and are supposed to be completely eliminated by the rewrite rules.

The approach is illustrated in Fig. 4, which shows how the rewrite system of Fig. 3 can be turned into a terminating rewrite system that defines the normalization to disjunctive normal form (DNF). To normalize a formula to DNF the function `dnf` should be applied to it. Normalization to conjunctive normal form requires a similar definition. The `dnf` function mimics the innermost normalization strategy by recursively traversing terms. The auxiliary functions `not` and `and` are used to apply the distribution rules and the negation rules. In functional programming such auxiliary functions are known as *smart constructors* (Elliot et al., 2000). In the definition of the rules for `and` and `not` it is assumed that the arguments of these functions are already in disjunctive normal form. For example, if none of the arguments of `and` is an `Or` term, the term itself is considered to be in DNF.

In the solution in Fig. 4, the original rules have been completely intertwined with the `dnf` transformation. The rules for negation cannot be reused in the definition of normalization to conjunctive normal form. For each new transformation a new traversal function and new smart constructors have to be defined. Many additional rules had to be added to traverse the term to find the places to apply the rules. Instead of 5 rules, a total of 13 rules were needed. Rules `AND3` and `NOT4` are default rules that only apply if the other rules do not apply. Without this mechanism even more rules would have had to be used to handle the cases were the real transformation rules do not apply.

The kind of problem illustrated in the example above occurs frequently in all kinds of transformations. In general, trying to overcome the problems of non-termination and non-confluence leads to encoding of control in terms of additional rewrite rules (which is at variance with our goal to separate rules from strategies as much as possible). This usually

```
imports integers
signature
  sorts Int
  constructors
    Fac : Int -> Int
    If  : Bool * Int * Int -> Int
rules
  Fac : Fac(x) -> If(Eq(x,0), 1, Mul(x,Fac(Sub(x,1))))
  IfT : If(True, x, y) -> x
  IfF : If(False, x, y) -> y
```

Fig. 5. Rewrite system with non-terminating reduction path.

leads to a functional programming style of rewriting, overhead in the form of traversal rules for each constructor in the signature, intertwining of rules and function definitions, all of which makes reuse of rules impossible, and leads to specifications that are much harder to understand.

4.6. *Conditional term rewriting*

Transformation rules often need more information than is provided by the match of the left-hand side to decide whether the rule is applicable. Thus, side conditions can be attached for checking additional properties. Such conditions may entail computations in a different paradigm than the transformation rules are implemented in. For example, in Sittampalam et al. (2004) *regular path expressions* checking data-flow properties of the program statement to be performed are attached to transformation rules (see Section 8).

In *conditional term rewriting* (Dershowitz and Jouannaud, 1990), however, the rewriting mechanism itself is used to evaluate conditions. Conditions are *equations* over terms, and a conditional rewrite rule has the form

$$t \texttt{ -> } t' \texttt{ where } t_1 = t_1' \texttt{ .... } t_n = t_n'\texttt{.}$$

When applying such a rule the equations are instantiated according to the substitution obtained from the match of the left-hand side. The pairs of terms are then compared for equality *after* rewriting them to normal form with the same set of rules.

In a variation on this concept, one side of a condition may use variables not occurring in the left-hand side of the rule. This term is then used to match the normal form of the other side of the equation against. The resulting variable bindings can be used in further conditions and the right-hand side of the rule.

4.7. *Term rewriting with strategy annotations*

One problem in term rewriting is that of terms with infinite reduction paths that cannot be resolved by removing unnecessary rules. For example, the specification in Fig. 5 defines the computation of the factorial function using the conditional `If`. Using a pure innermost rewriting strategy, a term `Fac(3)` does not terminate, since the arguments of `If` are evaluated before rules `IfF` or `IfT` are applied. While using an outermost strategy might solve termination problems like these, the cost of finding the next redex is much lower

```
signature
  sorts Nat List(*)
  constructors
    Z    : Nat
    S    : Nat -> Nat
    Cons : a * List(a) -> List(a) {strat: (1 0)}
    Inf  : Nat -> List(Nat)
    Nth  : List(a) -> a
rules
                 Inf(x) -> Cons(x, Inf(S(x)))
     Nth(Z, Cons(x, l)) -> x
  Nth(S(x), Cons(y, l)) -> Nth(x, l)
```

Fig. 6. Specification with strategy annotations (Ogata and Futatsugi, 1997).

in innermost rewriting (van de Pol, 2001). Therefore, several systems extend innermost rewriting with *strategy annotations* to delay the evaluation of arguments.

4.7.1. Just-in-time

The strategy annotations in van de Pol (2001) are designed to delay the evaluation of arguments, but guarantee that the term reached after evaluation is a normal form with respect to the rewrite system, i.e., contains no redices.

A strategy annotation for a constructor is a list of argument positions and rule names. The argument positions indicate the next argument to evaluate and the rule names indicate a rule to apply. The innermost strategy corresponds to an annotation `strat(C) = [1,2,3,...,R1,R2,R3,...]` for a constructor `C` and indicates that first all its arguments should be evaluated and then the rules `Ri` should be applied. By requiring that all argument positions and all rules for a constructor are mentioned in the annotation, it can be guaranteed that a normal form is reached. The just-in-time strategy is a permutation of argument positions and rules in which rules are applied as early as possible.

Using these annotations the non-termination for the rewrite system in Fig. 5 is solved by means of the annotation

```
strat(If) = [1,IfT,IfF,2,3]
```

that declares that only the first argument should be evaluated before applying rules `IfT` and `IfF`.

4.7.2. E-Strategy

The just-in-time strategy cannot deal with rewrite systems that do not have normal forms for some terms. For example, consider the rules in Fig. 6. Terms of the form `Inf(n)`, for some natural number `n`, do not have a normal form.

The *evaluation strategy* of the OBJ family of systems (Ogata and Futatsugi, 1997; Goguen et al., 2000) uses an extended form of strategy annotations in which not all arguments need to be evaluated. In this style a strategy annotation is a list of argument positions and the root position (`0`). The annotation declares the order of evaluation of the arguments. The root position `0` indicates the evaluation of the term at the root. Not all argument positions need to be declared. An undeclared argument is not evaluated.

```
rules
                   Inf(x) -> Cons(x, Thunk(L, Vec1(x)))
        Nth(Z, Cons(x, l)) -> x
     Nth(S(x), Cons(y, l)) -> Nth(x, Inst(l))
  Inst(Thunk(L, Vec1(x))) -> Inf(S(X))
                  Inst(x) -> x
```

Fig. 7. Result of translating specification with laziness annotations to eager specification (Fokkink et al., 2000).

For example, the non-termination in Fig. 6 is solved by the strategy annotation `(1 0)`, which indicates that first the first argument of `Cons` should be evaluated and then the constructor itself (`0`). The second argument is never evaluated. The E-normal form of `Nth(S(Z),Inf(Z))` is `S(Z)`. Also the term `Inf(Z)` has a normal form, i.e., `Cons(Z,Inf(S(Z)))`.

4.7.3. Laziness annotations

The strategy annotations discussed above are interpreted by the rewrite engine. In Fokkink et al. (2000) it is shown how rewrite systems with laziness annotations can be compiled into rewrite systems that can be evaluated using an innermost strategy.

A laziness annotation indicates for an argument of a constructor that it is lazy, i.e., that no reductions should be performed for subterms of that argument, unless needed for matching. For example, for the rewrite system in Fig. 6 the laziness annotation `Lazy(Cons,2)` achieves the delay of the evaluation of the second argument of `Cons`.

A rewrite system with laziness annotations can be translated to an eager rewrite system using thunks. A thunk is an auxiliary data structure that stores the structure of the term. For example, the term rewrite system (TRS) in Fig. 6 is transformed to the eager TRS in Fig. 7. Note that `Thunk` is a generic constructor for representing thunks, `L` is a constructor for indicating the thunked pattern, and `Vec1` is a constructor for denoting a vector of length 1.

Note that annotations depend on the application in which they are used. For example, without the `Inf` constructor there is no reason for annotating the second argument of `Cons` as lazy.

5. Tree parsing strategies

Tree parsing is an alternative approach to transformation developed in the area of code generation. In this approach rules are written as *tree grammar rules* that are used to *parse* a tree, i.e., cover the tree with applicable rules and execute corresponding actions. This requires deriving from the specification of the rules a tree traversal schedule. This section discusses three approaches to tree parsing. Simple tree parsing is used to generate single-pass traversals. Bottom-up tree parsers are used in code generators and employ a dynamic programming approach to compute all possible rewrites in parallel. Finally, in *attribute grammars* rules assign attribute values to tree nodes; attribute evaluation involves *scheduling* of the order of evaluation based on dependencies.

5.1. Tree parsing

Tree parsing is analogous to string parsing; instead of finding structure in a string, the goal is to find structure in a tree by covering the tree with patterns. The tree parser generator for the ANTLR language processing system (Parr et al., 2003) generates tree walkers from tree grammars. A tree grammar is a BNF-like notation for the definition of tree structures. For example, the grammar

```
exp : #(PLUS exp exp)
    | INT
```

describes expression trees composed from integers and addition.

Tree translations and transformations are achieved by associating actions with the grammar productions. Translations to textual output are achieved by printing actions. For example, the following grammar prints expressions using infix notation.

```
exp : #(PLUS exp <<printf("+");>> exp)
    | i:INT <<printf("%d", i);>>
```

Tree transformations are achieved by reconstructing trees and returning them as results. For example, the following grammar transforms expressions by swapping the arguments of the `PLUS` operator.

```
exp :! #(PLUS l:exp r:exp) <<#exp = #(PLUS r l);>>
    | INT
```

Grammar non-terminals can have arguments that can be used in the actions in productions. Non-terminals can also return results. A tree grammar gives rise to a set of mutually recursive functions, one for each non-terminal, that together define a one-pass traversal over a tree. Patterns can be nested and can use regular tree expressions with optionals, alternatives and lists.

Transformation rules in tree grammars are embedded in grammar productions. Separation of rules and strategies and generic tree traversals are not supported in ANTLR.

5.2. Bottom-up tree parsing

If a tree grammar is ambiguous, multiple parses of a tree are possible. The parser needs to decide which parse to take. By associating costs with each production, the disambiguation can be based on the accumulated cost of a tree. Dynamic programming techniques can be used to compute all possible parses in one traversal.

BURG (Fraser et al., 1992a,b; Proebsting, 1995) is a system for code generation from intermediate representation (IR) expression trees. A mapping from IR trees to machine instructions is defined by means of a tree grammar. A production of the form `n -> t (c)` defines a mapping of tree pattern `t` to non-terminal `n` at cost `c`. Associated with each production is an action to take when the production is selected. For example, (Proebsting, 1995) gives the example grammar in Fig. 8. According to this grammar, the term `Fetch(Fetch(Plus(Reg,Int)))` has two coverings corresponding to the derivations `4(4(6(5(2,3))))` and `4(4(8(2)))` with costs 7 and 4, respectively.

```
[1] goal -> reg         (0)    [5] reg  -> Plus(reg, reg) (2)
[2] reg  -> Reg         (0)    [6] addr -> reg            (0)
[3] reg  -> Int         (1)    [7] addr -> Int            (0)
[4] reg  -> Fetch(addr) (2)    [8] addr -> Plus(reg, Int) (0)
```

Fig. 8. Example BURG specification.

As illustrated by this example, more than one covering of a tree is possible, corresponding to different ways to generate code. Each node can have several different parses because of overlapping patterns and chain rules. The costs associated with the productions express the cost of executing the associated machine instruction. The goal of a code generator is to find the lowest cost covering (i.e., lowest execution time) of an intermediate representation expression tree.

According to bottom-up rewriting theory (BURS) an IR tree can be translated to a sequence of instructions using the following strategy. In a bottom-up traversal all lowest-cost patterns that match each node are computed and associated with the node. This involves matching the right-hand sides of the productions to the tree, taking into account earlier matches for subtrees. Instructions are then selected in a top-down traversal that is driven by the goal non-terminal for the root of the tree.

This restricted form of rewriting can also be applied (Proebsting, 1995) for simple type inference problems, for checking tree formats, and for tree simplifications. However, the scope of this paradigm is restricted to one-to-one translations in which the structure of the target is closely related to the structure of the source program, which is typically the case in instruction selection.

5.3. Attribute grammars

Attribute grammars (Knuth, 1968; Aho et al., 1986) provide a more general form of tree parsing. Instead of associating actions with a fixed tree traversal, an attribute grammar defines the computation of *attribute values* associated with tree nodes. Typically, the values of an attribute can be defined in terms of the values of other attributes. Thus tree traversal is implicit, i.e., inferred from the attribute definition rules.

As an example, consider the following set of rules defining the computation of the set of free variables of a lambda expression with variables (Var), lambda abstraction (Abs), and application (App):

```
e@Var(x)        : e.free  := [x]
e0@Abs(x, e1)   : e0.free := <diff>(e1.free, [x])
e0@App(e1, e2)  : e0.free := <union>(e1.free, e2.free)
```

The grammar consists of productions associating attribute evaluation rules with tree constructors. The rules refer to the current node or its direct subnodes via identifiers. The example grammar defines the attribute `free`, which evaluates to the set of free variables of an expression. These attributes are so-called *synthesized* attributes, since the attribute value of a node is defined in terms of the attribute values of the subnodes. This example illustrates how attribute grammars can be used for *analysis*.

Higher-order attribute grammars (Vogt, 1989) can also be used for *transformation* by computing new trees as part of attribute evaluation. The following example illustrates this in a grammar for renaming bound variables in lambda expressions:

```
e@Var(x)        : e.rn      := Var(<lookup>(x, e.env))
e0@Abs(x, e1)   : e0.rn     := Abs(y, e1.rn)
                  e1.env    := [(x,y) | e0.env]
                  y         := <first> e0.ifresh
                  e1.ifresh := <next> e0.ifresh
                  e0.sfresh := e1.sfresh
e0@App(e1, e2)  : e0.rn     := App(e1.rn, e2.rn)
                  e1.env    := e0.env
                  e2.env    := e0.env
                  e1.ifresh := e0.ifresh
                  e2.ifresh := e1.sfresh
                  e0.sfresh := e2.sfresh
```

This grammar defines the synthesized attribute `e.rn`, which evaluates to the renaming of the lambda expression `e`. Two auxiliary attributes are used in the definition. The attribute `e.env` is an *inherited* attribute, which is passed to subnodes and maintains the mapping from variables to their new name. The attribute `e.ifresh` is also an inherited attribute providing a supply of fresh names. The synthesized attribute `e.sfresh` evaluates to the state of the name supply after computing the renaming of `e`. This is used to *thread* the name supply through the computation.

Attribute evaluation requires dependency analysis in order to determine a traversal schedule. Such a schedule may involve multiple traversals over the tree when an inherited attribute depends on a synthesized attribute, which should thus be computed in an earlier traversal. Attribute grammars are used in various systems such as the Synthesizer Generator (Reps and Teitelbaum, 1988) and LRC (Saraiva and Kuiper, 1998). In the intentional programming project (Czarnecki and Eisenecker, 2000) attribute grammars are used for the definition of language extensions. Attribute grammars have been especially successful in *incremental* computation of attribute values, enabling rapid feedback in an interactive environment (Reps and Teitelbaum, 1988). Explicit scheduling of attribute evaluation is not necessary when implementing attribute grammars in a lazy functional language (Johnsson, 1987). Scheduling is achieved by the evaluation mechanism of the host language. This is exploited in systems such as Elegant (Augusteijn, 1993) and UUAG (Saraiva and Swierstra, 1999).

The example above illustrates that attribute propagation requires numerous administrative rules. In extensions of the basic formalism, reoccurring patterns such as *broadcasting*, *threading*, and *collecting* values are provided through declarations. In the proposal for *first-class attribute grammars* in de Moor et al. (2000), such patterns are *programmable*.

6. Programmable strategies

Systems with fixed strategies are developed for application in specific domains such as instruction selection, but are not sufficient as general purpose transformation systems.

The inappropriateness of the standard strategy to a specific application, invariably leads to encoding of control in rules. In previous sections we saw various extensions to rule-based systems that allowed *adaptations* of the standard strategy; however, no completely different strategies are allowed in such settings. In the areas of transformational programming and theorem proving the tediousness of the interactive application of rules required more automation, while retaining control. This led to the extension of rule-based systems with *tactics*, i.e., specific algorithms for applying rules. These systems influenced the design of program transformation systems with *programmable strategies*, i.e., providing a basic set of combinators from which complex strategies can be composed. This section sketches the development from interactive program transformation systems to systems for automatic program transformation with programmable strategies. The next section then focuses on a particular aspect of such strategies, namely the specification of *term traversal*.

6.1. Interactive program transformation

The *transformational* approach to software development is based on the paradigm of *top-down stepwise refinement* (Dijkstra, 1968) in which a high-level specification of a problem is gradually refined to an efficient implementation solving that problem. The aim is to achieve orders of magnitude improvement in programmer productivity (Paige, 1994). By only applying *correctness-preserving* transformations, the resulting program is *correct by construction*. Transformation from high-level specification to low-level implementation gives rise to *wide spectrum languages* containing constructs for very high-level specification (e.g., non-executable logic formulae) as well as low-level implementation (e.g., assembly language instructions). Thus, all transformations are performed on the same language. An alternative approach, pioneered in the Draco system (Neighbors, 1984), is to define *domain-specific languages* that only cover a specific application domain and level of abstraction, thus limiting the complexity of transformations.

First of all the approach required the development of theories for program transformation (Burstall and Darlington, 1977; Bird and Meertens, 1987) consisting of basic rules such as *fold* and *unfold* and strategies such as *composition*, *generalization*, and *tupling* applying these rules in a certain combination (Pettorossi and Proietti, 1996b). Using such a theory, programs can be derived mechanically from specifications. Since manual application of rules is tedious and error-prone, automation was a logical course. Thus, over the last 30 years many systems have been developed to support some variation on the transformational approach. Examples include ZAP (Feather, 1982), Programmer's Apprentice (Waters, 1982; Rich and Waters, 1990), Draco (Neighbors, 1984), KIDS (Smith, 1990), CIP (Partsch, 1990), APTS (Paige, 1994), Map (Pettorossi and Proietti, 1996b), and Ultra (Partsch et al., 1999; Guttmann et al., 2003). Although the systems differ in the details of their implementation, the kinds of transformation they apply, and the languages that are transformed, they have many commonalities. First of all the systems are usually specific for a programming language. On the basis of the semantics of this language a library of valid and usually generic transformation rules is developed. Since the declarative nature of the specifications allows many design choices, the systems are interactive and let the user select the rules to apply and the order in which to apply them. Thus, a basic transformation assistant is an aid to

do the bookkeeping for applying rules. Since the complete development of a program may require the application of hundreds or thousands of rules, assistance in applying single rules is not sufficient, and various mechanisms (*tactics*) for automatically applying certain combinations of rules were added. Thus, instead of a fixed (exhaustive) strategy, these systems allow the programmer to *choose* the strategy to apply certain rules with.

The transformational programming approach has reincarnated in a different guise in the area of software maintenance. *Refactoring* aims at cleaning up the design of the code rather than producing a performance improvement (Fowler, 1999). Refactoring editors such as the Smalltalk Refactoring Browser (Roberts et al., 1997) are the modern incarnation of the programmer's assistant. That is, they are interactive tools that allow the programmer to apply specific transformation rules to selected parts of the program. Typically, however, refactoring rules are more coarse grained than single-fold/unfold rules. Work so far concentrates on the implementation of specific refactorings. If this line of work turns out to be successful, refactoring editors will be extended with scripting capabilities in order to combine refactorings into more complex refactorings.

6.2. Staging

In transformational programming a transformation is geared to the transformation of one specific program, possibly employing reusable transformation tactics. In *automatic* program transformation used in compilers, for example, programmer intervention is not desirable since it is not reproducible and costs a lot of time. As argued before, pure rewriting is not applicable because of interference between rewrite rules. A step towards programmer control over rules is the mechanism of *sequence of canonical forms* adopted in the TAMPR — Transformation Assisted Multiple Program Realization — system (Boyle, 1989; Boyle et al., 1997), aiming to derive implementations for different computer architectures from the same specification, in particular in the domain of numerical programming.

A TAMPR specification consists of a series of rewrite rules. The TAMPR rewrite engine applies rewrite rules exhaustively to reach a canonical form. The problem of non-termination caused by rules that are each others' inverses is solved by organizing a large transformation into a sequence of consecutive reductions to canonical forms under different sets of rewrite rules. Typically such a sequence starts with several preparatory steps that bring the program into the right form, and these are followed by the pivotal step which achieves the actual transformation, followed in turn by some post-processing.

As an example consider the transformation of nested function calls to flat lists of function calls as part of a compiler from a functional program to an imperative program (inspired by an example in Boyle (1989)). The following pair of program fragments illustrates the transformation:

```
let var x := foo(bar(a, b))
 in ... end
```

⇒

```
let var y := bar(a, b)
 in let var x := foo(y)
     in ... end
end
```

In the canonical form that is reached by the transformation, each function call is directly assigned to a variable and has no nested function calls. The transformation is achieved

```
rules
  IntroduceTemp :
    |[ f(a*) ]| -> |[ let var x := f(a*) in x end ]|
    where new => x
  LetFromApp :
    |[ f(a1*, let var x := e1 in e2 end, a3*) ]| ->
    |[ let var x := e1 in f(a1*, e2, a3*) end ]|
  LetVarInLetVar :
    |[ let var y := let var x := e1 in x end in e2 end ]| ->
    |[ let var y := e1 in e2 end ]|
  LetFromLet :
    |[ let var y := let var x := e1 in e2 end in e3 end ]| ->
    |[ let var x := e1 in let var y := e2 in e3 end end ]|
    where <not(eq)>(|[ x ]|, |[ e2 ]|)
strategies
  lift-nested-calls =
    one-shot(IntroduceTemp)
    ; transform*({LetFromApp, LetVarInLetVar, LetFromLet})
```

Fig. 9. Rewrite rules and strategy for lifting nested function calls.

by a number of simple rewrite rules (Fig. 9) that first assign each function call to a fresh variable by introducing a new let binding. Then these bindings are lifted by distribution rules pushing function applications and let bindings inside the body of the nested let binding. Since rule `IntroduceTemp` is a non-terminating rule, some mechanism is needed to control its application. The TAMPR approach is to organize a transformation as a sequence of exhaustive normalizations and one-shot rule applications. Thus, the call lifting transformation is defined by the strategy `lift-nested-calls`, which first tries to apply rule `IntroduceTemp` exactly once to all nodes in the program, and then exhaustively applies the other rules.

In Fitzpatrick et al. (1995) the authors state that: *A major issue still to be addressed in transformation systems is the control of the derivation process; i.e., the specification of* strategies *to achieve some goal.* The division of a rewrite systems into separate sets of rules which are applied exhaustively in sequence does solve some of the termination and confluence problems of rewriting, and it nicely preserves the declarative nature of individual rewrite rules. However, many problems need to be addressed by other means within a single traversal. For such cases, the TAMPR approach still requires the use of functional rewriting.

6.3. Strategy combinators

Taking the approach of TAMPR further requires more expressive specification of strategies to control rule application, while preserving the separation of rules and strategies. The algebraic specification language ELAN (Borovanský et al., 2000, 1996, 2002, 1998) introduced support for *user-definable* strategies using a language of combinators for

```
strategies
  try(s)           = s <+ id
  repeat(s)        = try(s; repeat(s))
  while(c, s)      = try(c; s; while(c, s))
  do-while(s, c)   = s; try(c; do-while(s, c))
  while-not(c, s) = c <+ s; while-not(c, s)
  for(i, c, s)     = i; while-not(c, s)
```

Fig. 10. Iteration strategies defined using strategy combinators.

composing strategies. The approach was also adopted in the design of the program transformation language Stratego (Visser et al., 1998; Visser, 2004). Despite differences in syntax and the sets of combinators, the basic ideas underlying the strategy combinators of ELAN and Stratego are the same. Here the ideas of the approach are explained using the Stratego syntax. Where there are real differences, these will be pointed out explicitly.

In the *strategic rewriting* approach, a specification consists of a set of *labeled rewrite rules* and a set of *strategy definitions*. Strategies are programs that attempt to transform terms into terms, at which they may succeed or fail. In the case of success the result of such an attempt is a transformed term. In the case of failure there is no result of the transformation. The *atomic* strategies are the labels of rewrite rules, the *identity* strategy `id`, which leaves the subject term unchanged and always succeeds, and the *failure* strategy `fail`, which always fails. These atomic strategies can be combined into more complex strategies by means of a set of strategy *combinators*. The *sequential composition* `s1 ; s2` of strategies `s1` and `s2` first attempts to apply `s1` to the subject term and, if that succeeds, applies `s2` to the result. The *non-deterministic choice* `s1 + s2` of strategies `s1` and `s2` attempts to apply either `s1` or `s2`. It succeeds if either succeeds and it fails if both fail; the order in which `s1` and `s2` are tried is unspecified. The *deterministic choice* `s1 <+ s2` of strategies `s1` and `s2` attempts to apply either `s1` or `s2`, in that order. Note that `;` has higher precedence than `+` and `<+`. The *test* strategy `test(s)` tries to apply the strategy `s`. It succeeds if `s` succeeds, and reverts the subject term to the original term. It fails if `s` fails. The *negation* `not(s)` succeeds (with the identity transformation) if `s` fails and fails if `s` succeeds. A recursive strategy can be defined using a recursive definition.

As an example of the versatility of these basic combinators, Fig. 10 defines a number of derived control combinators, corresponding to various iteration schemes. To illustrate the use of these strategies consider again the rules for evaluation of the `Member` function:

```
rules
  Mem1 : Member(x, [])        -> False
  Mem2 : Member(x, [x | xs]) -> True
  Mem3 : Member(x, [y | xs]) -> Member(x, xs)
strategies
  member = repeat(Mem1 <+ Mem2 <+ Mem3)
```

In Section 4.4, the order of the application of these rules was enforced by making `Mem3`, a *default* rule. Using the strategy combinators introduced above, the priority between rules can be explicitly stated. Thus, the `member` strategy repeatedly applies rules `Mem1`, `Mem2`, and `Mem3` *in that order*.

Note that labeled rules apply at the root of the term to which they are applied. Furthermore, the combinators introduced above also apply to the root and do not descend into the term. This makes it impossible to apply rules below the root. In ELAN this is remedied by means of two devices. First, using so-called *congruence operators* traversals over a tree can be defined. This device will be further discussed in the next section. Second the special strategy `normalize(`$L_1, \ldots, L_n$`)` normalizes a term with respect to rules $L_1, \ldots, L_n$. Thus, the normalizing bit of the call lifting strategy can be defined as

```
normalize(LetFromApp,LetVarInLetVar,LetFromLet)
```

The one-shot strategy requires a special traversal, which will be discussed in the next section.

Another difference between ELAN and Stratego is the *backtracking* model. The choice operators + and <+ of Stratego provide *local* backtracking. This means that the choice is committed after a successful alternative has been applied. Thus in the strategy $(x + y); z$ if x is tried first and succeeds the choice is committed and z is applied. If z fails the entire strategy fails instead of backtracking to y, which might potentially succeed and make z succeed as well. Thus, $(x + y); z$ is not equal to $(x; z + y; z)$.

In ELAN there are several choice operators each with different backtracking properties. The failure/success model is based on *sets of results*, i.e., a strategy returns a set of possible results, which can be implemented using *global backtracking*. That is, at a successful choice, the remaining choices are stored in case a continuation strategy fails. The operator `dk(e1,...,en)` (don't know) returns all results from all strategies `ei`. The operator `dc(e1,...,en)` (don't care) returns the results from one of its argument strategies as long as it does not fail. The operator `first(e1,...,en)` returns the results of the first `ei` that does not fail. The operators `dc_one(e1, ..., en)` and `first_one(e1, ..., en)` return only one result. The operator `iterate*(e)` (respectively, `iterate+(e)`) returns all possible results from iterating the strategy `e` zero (respectively, one) or more times. The operator `repeat*(e)` (`repeat+(e)`) returns the last set of results from repeatedly applying `e` until it fails.

Finally, ELAN also has unlabeled rewrite rules, which are always applied using a fixed innermost strategy, i.e., not under the control of a strategy. Another feature of ELAN (Borovanský et al., 1996) is the reflective rewriting of strategies with rewrite rules, which is possible since the strategy language is interpreted. ELAN does not support generic term traversal, a feature to be discussed in the next section.

7. Traversal strategies

A special concern in any implementation of program transformation is the definition of *traversals* that determine the order in which the nodes of an abstract syntax tree are visited. In the pure rewriting approach traversal is implicit in the strategy. However, we saw in Section 4.5 that rewriting often degrades to functional programming with a steep penalty for the definition of traversals. This penalty is especially large in program transformation, where languages with tens to hundreds of constructors are common. Definition of a traversal for *each* transformation to be defined leads to very large specifications. There are several approaches to solving this problem, which will be discussed in this section.

```
signature
  constructors
    dnf : Prop -> Prop {traversal(trafo,bottom-up,continue)}
    and : Prop * Prop -> Prop
    not : Prop -> Prop
rules
  DNF4  : dnf(Not(x))     -> not(x)
  DNF5  : dnf(And(x,y))   -> and(x,y)

  AND1  : and(Or(x,y),z) -> Or(and(x,z),and(y,z))
  AND2  : and(z,Or(x,y)) -> Or(and(z,x),and(z,y))
  AND3  : and(x,y)        -> And(x,y) (default)

  NOT1  : not(Not(x))     -> x
  NOT2  : not(And(x,y))   -> Or(not(x),not(y))
  NOT3  : not(Or(x,y))    -> and(not(x),not(y))
  NOT4  : not(x)          -> Not(x) (default)
```

Fig. 11. Disjunctive normal form with traversal function (Version 1).

7.1. Traversal functions

In ASF+SDF controlling the application of transformation rules has been recognized as a problem for a long time. For the specification of transformations for large languages such as COBOL the overhead of defining traversals was seen as an especially problematic factor. First this was solved by the generation of default traversal rules (van den Brand and Visser, 1996; van den Brand et al., 2000b) that could be overridden by normal rules. In this approach typically only a few rewrite rules have to be specified, corresponding to the non-default behavior of the traversal. However, the number of generated rules still proves to be a source of overhead, albeit for the compiler, not the programmer. Furthermore, providing a new traversal scheme requires the addition of a new generator.

In a recent approach (van den Brand et al., 2003), traversal functions are supported directly by the rewriting engine, avoiding the compile-time overhead of generated rules. The transformation language TXL (Cordy et al., 1995) provides a similar approach. Fig. 11 illustrates the approach applied to the problem of normalization to disjunctive normal form. The specification is the same as that in Fig. 4, but the `dnf` function has been declared a traversal function in the signature. The attribute `traversal(trafo, bottom-up, continue)` declares that `dnf` performs a bottom-up traversal over its argument. This means that the function is first applied to the direct subterms (and, thus, recursively to all subterms) before it is applied at the term itself. Rules need to be declared only for those constructs that are transformed. The default behavior is to reconstruct the term with the original constructor. In the example this reduces the specification of the traversal from six to two rules. In general, for a signature with n constructors only m of which need to be handled in a special way, this saves $n - m$ rules.

There is still some overhead in the specification in Fig. 11 in the form of the dispatching from the traversal function to the smart constructors and the default rules for the smart constructors. A more concise specification is the one in Fig. 12 in which no smart constructors are used. In this style only one rule is needed for each original rule. However,

```
signature
  constructors
    dnf : Prop -> Prop {traversal(trafo,bottom-up,continue)}
rules
  AND1  : dnf(And(Or(x,y),z) -> dnf(Or(And(x,z)),And(y,z))
  AND2  : dnf(And(z,Or(x,y)) -> dnf(Or(And(z,x)),And(z,y))

  NOT1  : dnf(Not(Not(x))    -> x
  NOT2  : dnf(Not(And(x,y))  -> dnf(Or(Not(x),Not(y)))
  NOT3  : dnf(Not(Or(x,y))   -> dnf(And(Not(x),Not(y)))
```

Fig. 12. Disjunctive normal form with traversal function (Version 2).

the problem with this style is that the recursive calls in the right-hand sides of the rules will completely retraverse the tree (the arguments of which are already normalized) before applying one of the rules.

The traversal strategy of a traversal function is based on choices in several aspects of a traversal. First of all a traversal can be a *transformation* (`trafo`) that changes the shape of a tree and/or a *accumulator* (`accu`) that collects information during the traversal. Secondly the *node visiting order* of a traversal can be either `top-down` and `bottom-up`. Finally, the traversal can stop as soon as a rule has been applied successfully (`break`), or can continue (`continue`).

The advantage of traversal functions is that default traversal behavior does not need to be implemented manually. This is similar to the case of default visitors in object-oriented programming or folds with updatable fold algebras in functional programming. However, the approach has a number of limitations.

First of all, there is no separation of rules from strategies. A rule is bound to one specific traversal via the traversal function. It is not possible to reuse rules in different traversals, for example, to normalize under different rule sets. Furthermore, rules are intertwined with strategies, making it hard to distinguish the basic transformation rules from the traversal code.

Secondly, although it is possible to implement a wide range of traversals, this requires gluing together the basic traversals in an ad hoc manner. That is, traversal schemata are not first-class citizens of ASF+SDF. It is not possible in the language to give further abstractions for alternative or composite traversal schemata, or for more elaborate functionality involving traversals. That would require extending the rewriting engine interpreter and compiler. Such extensibility is desirable for building libraries with language independent strategies. For example, defining substitution without variable capture is similar for many languages, given the shape of variables and variable bindings. Extrapolating the traversal function approach, more and more such abstractions will be captured as additional primitives in the rewrite engine. At some point it will make sense to extend the language with a mechanism for specifying such abstractions generically.

7.2. *Folds*

Folds or *catamorphisms* in functional languages are an approach to traversal that does admit reuse and definition of new traversal schemes. Instead of redefining the traversal

for each transformation, a higher-order definition of a traversal is defined, which can be instantiated for different applications. For example, the fold for lists is defined as

```
foldr(n, c) : [] -> n
foldr(n, c) : [x | xs] -> <n>(x, <foldr(n, c)> xs)
```

and can be used for the definition of the `map` function, which applies a function `f` to each element of the list:

```
map(f) = foldr([], \ (x, xs) -> [<f>x | xs] )
```

Thus, a fold performs a bottom-up traversal applying to the result of traversing the subnodes, a function corresponding to the original constructor.

This idea can be generalized to arbitrary data types. For example, the fold for lambda expressions is defined as

```
foldexp(var, app, abs) :
  Var(x) -> <var> x
foldexp(var, app, abs) :
  App(e1, e2) -> <app>(<foldexp(var, app, abs)> e1,
                       <foldexp(var, app, abs)> e2)
foldexp(var, app, abs) :
  Abs(x, e) -> <abs>(x, <foldexp(var, app, abs)> e)
```

This function can be used in the definition of free variable extraction, for example

```
free-vars =
  foldexp(id, union, \ (x,xs) -> <diff>(xs, x))
```

However, it is not usable for bound variable renaming, since no information is passed down the tree.

There are other shortcomings as well. Folds and similar traversals define a full traversal over the tree. It is not always appropriate to apply a transformation uniformly to the *entire* tree. Furthermore, the function is parameterized with a function *for each* constructor. This not feasible for realistic abstract syntax trees with tens or hundreds of constructors, since it requires the specification of replacement functions for each constructor in the signature of the data type. Updatable fold algebras (Lämmel et al., 2000) are an attempt at mitigating the number of parameters by storing the constructor functions in a record.

7.3. Traversal with congruence operators

Congruence operators, introduced in ELAN and adopted by Stratego, provide more fine grained primitives for composing traversals. For each `n`-ary constructor `C` a *congruence* strategy operator of the form `C(s1,...,sn)` is available. It applies to terms of the form `C(t1,...,tn)`, applying each `si` to the corresponding `ti`. An example of the use of congruences is the operator `map(s)`

```
map(s) = [] + [s | map(s)]
```

which applies a strategy `s` to each element of a list.

Congruence operators are very useful for defining traversals that are *specific* for a language. This is used for example in the definition of a traversal that follows the control flow in an interpreter (Dolstra and Visser, 2002). For example, given rules Beta for beta-reduction, and SubsVar, SubsApp, and SubsAbs for substitution, the strategy

```
eager-eval = rec e(
  try(App(e, e) + Let(id, e, e))
  ; try((Beta + SubsVar + SubsApp + SubsAbs); e)
)
```

defines eager evaluation for lambda expressions, i.e., that in which inner redices are reduced first, but where no reduction under lambdas is performed. A variation on this strategy is

```
lazy-eval = rec e(
  try(App(e, id) + Let(id, id, e))
  ; try((Beta + SubsVar + SubsApp + SubsAbs); e)
)
```

which defines *lazy* evaluation, i.e., where no reductions of function *arguments* are performed. This approach is also used in the specification of data-flow optimizations (Olmos and Visser, 2002) and partial evaluation (Olmos and Visser, 2003).

Another application of congruence operators is in the definition of *format checkers*, used to check syntactic properties of terms. For instance, the following definitions

```
conj(s) = And(conj(s),conj(s)) <+ s
disj(s) = Or(disj(s),disj(s)) <+ s
dnf     = disj(conj(Atom(id) + Not(Atom(id))))
```

define the strategy dnf, which checks that a propositional formula is in disjunctive normal form (Visser, 1999).

The difference between folds and congruence operators is that the former define a complete recursive traversal over a tree, whereas the latter define only a one level descent into the subtrees. This entails that different traversals can be composed from the same basic building blocks. Similarly to the folds case, however, it is still necessary to use the congruence operators for all constructors which should be traversed.

7.4. *Generic traversal strategies*

The approaches to traversal discussed previously in this section all improve some aspect of traversal specification, but have shortcomings as well. Traversal functions are generic in the tree structure, but they are whole tree traversals and do not admit definitions of new traversal schemata. Folds are parametric, but not generic in the tree structure and define whole tree traversals. Congruence operators are fine grained, i.e., partial tree traversals, but are not generic in the tree structure, hence only reusable for a specific language. The solution to traversal introduced in Stratego (Luttik and Visser, 1997; Visser et al., 1998; Visser, 2004) combines the advantages of these approaches. The key to this solution is the notion of a *generic one-level traversal operator*, which can be used to freely compose many different *generic traversal strategies* (Luttik and Visser, 1997).

```
strategies
  topdown(s)      = s; all(topdown(s))
  bottomup(s)     = all(bottomup(s)); s
  downup(s)       = s; all(downup(s)); s
  downup2(s1,s2) = s1; all(downup2(s1,s2)); s2
  oncetd(s)       = s <+ one(oncetd(s))
  onecbu(s)       = one(oncebu(s)) <+ s
  alltd(s)        = s <+ all(alltd(s))
  sometd(s)       = s <+ some(sometd(s))
  somebu(s)       = some(somebu(s)) <+ s
  innermost(s)    = bottomup(try(s; innermost(s)))
```

Fig. 13. Generic traversal strategies.

A generic one-level traversal operator is similar to a congruence operator, which applies strategies to the immediate subterms of a term. The difference is that generic traversal operators are indifferent to the constructor of the term and uniformly apply a strategy to one or more of the subterms. For example, the traversal operator `all(s)` applies `s` to *all direct subterms* of a constructor application `C(t1,...,tn)`. The application succeeds with a new term `C(t1',...,tn')` constructed using the same constructor and the results of transforming the subterms with the strategy `s`, if those transformations succeed. Otherwise, the application fails. Similar one-level traversal operators are `one` and `some`. The strategy `one(s)` applies `s` to *one* direct subterm of a constructor application `C(t1,...,tn)`. The strategy `some(s)` applies `s` to some of the direct subterms of a constructor application `C(t1,...,tn)`, i.e., to at least one and as many as possible.

The genericity and fine granularity of these operators makes it possible to define a wide range of generic full traversals. For example, consider the traversal definitions in Fig. 13. The strategy `topdown(s)` defines a pre-order traversal visiting terms before descending to its subterms. The strategy `bottomup(s)` defines a post-order traversal, visiting a term after visiting its subterms. The strategy `downup(s)` visits subterms on the way down and on the way up. The strategy `oncetd(s)` tries to find *one* application of `s` somewhere in the term starting at the root working its way down; `s <+ one(oncetd(s))` first attempts to apply `s`, and if that fails an application of `s` is (recursively) attempted at one of the children of the subject term. If no application is found the traversal fails. The traversal `alltd(s)` finds *all* outermost applications of `s` and never fails.

These generic traversal strategies are parameterized with the actual transformation to be applied to the subterms. Fig. 14 gives several examples of uses of the strategies of Fig. 13. The strategies `disj-nf` and `conj-nf` define normalizations to disjunctive and conjunctive normal form, respectively, using the rules from Fig. 3. The `eval` strategy performs constant folding on propositional formulae using the standard truth rules `T` (not shown here). The strategies `desugar` and `impl-nf` define two desugarings of propositional formulae, i.e., elimination of implication and equivalence, and desugaring to implicative normal form using standard elimination rules (not shown here). These definitions illustrate how rules from the same collection can be reused in different transformations, and likewise, a generic strategy such as `innermost` can be instantiated to compose different transformations.

```
rules
  T : And(True, x) -> x ...
  T : Or(True, x) -> False ...
  DefI : Impl(x, y) -> Or(Not(x), y) ...
strategies
  disj-nf = innermost(DAOL + DAOR + DN + DMA + DMO)
  conj-nf = innermost(DOAL + DOAR + DN + DMA + DMO)
  eval    = bottomup(repeat(T))
  desugar = topdown(try(DefI + DefE))
  impl-nf = topdown(repeat(DefN + DefA2 + DefO1 + DefE))
```

Fig. 14. Various transformations on propositional formulae.

Using strategy combinators with one-level traversal operators, highly generic strategies can be defined. The Stratego library defines a wide range of generic strategies including the traversal strategies in Fig. 13. In addition the library defines a number of higher-level language-independent operations such as free variable collection, bound variable renaming, capture-free substitution, syntactic unification, and computing the spanning tree of a graph. These operations are parameterized with the relevant language constructs, but work generically otherwise (Visser, 2000).

Traversals can be combined in any way necessary. For example, the nested function call lifting strategy from Section 6.2 is defined as

```
lift-nested-calls =
  bottomup(try(IntroduceTemp))
  ; innermost(LetFromApp <+ LetVarInLetVar <+ LetFromLet)
```

where the one-shot strategy corresponds to a one-pass bottom-up traversal and normalization to canonical form is implemented with `innermost`.

A problem of some generic strategies is that they lack knowledge of the computations in their argument strategies, which may cause overhead. For example, the innermost strategy in Fig. 13 renormalizes arguments of left-hand sides of rules when they are used in the right-hand side. In Johann and Visser (2001) it is shown how this can be repaired by fusing the generic innermost strategy with its arguments.

The approach of generic traversal based on one-level descent operators has been adopted in Prolog (Lämmel and Riedewald, 2001), Haskell (Lämmel and Visser, 2002; Lämmel and Peyton Jones, 2003), and Java (Visser, 2001). An overview of the *strategic programming* approach is described in Lämmel et al. (2002). A comparison of this approach with *adaptive programming* is given in Lämmel et al. (2003). Static typing is an issue in a language with generic traversal. Solutions to this problem are explored in Lämmel (2003) for the setting of rewriting strategies, and in Lämmel and Visser (2002) for functional programming.

8. Context-sensitive rules

Another problem of rewriting is the context-free nature of rewrite rules. A rule has only knowledge of the construct it is transforming. However, transformation problems are often

context-sensitive. For example, when inlining a function at a call site, the call is replaced by the body of the function in which the actual parameters have been substituted for the formal parameters. This requires that the formal parameters and the body of the function are known at the call site, but these are only available higher up in the syntax tree. There are many similar problems in program transformation, such as bound variable renaming, typechecking, constant and copy propagation, and dead code elimination. Although the basic transformations in all these applications can be expressed by means of rewrite rules, they need contextual information. This section explores solutions in this area.

8.1. Parameterized strategies

The usual solution to this problem is to extend the traversal over the tree (be it hand-written or generic) such that it distributes the data needed by transformation rules. For example, traversal functions in ASF+SDF (van den Brand et al., 2003) can be declared to have an accumulation parameter in which data can be collected. Language independent definitions of operations such as bound variable renaming in Stratego (Visser, 2000) capture a generic tree traversal schema that takes care of distributing an environment through a tree. The disadvantage of these solutions is that the traversal strategy becomes data heavy instead of just handling control flow. That is, all traversal functions become infected with additional parameters carrying context information. Generic solutions break down when multiple environments are needed, to handle multiple name spaces, for instance.

8.2. Contextual rules

Another solution is the use of contextual rules (Appel and Jim, 1997; Visser et al., 1998). A contextual rule contains *context variables* of the form `e1[e2]` indicating an expression `e1` *containing* an occurrence of another expression `e2`. This allows replacing terms deeply nested in a term structure. For example, the rule

```
InlineVar :
  |[ let var x := e1 in e2[x] end ]| ->
  |[ let var x := e1 in e2[e1] end ]|
```

expresses the substitution of an occurrence in `e2` of a let-bound variable `x` with its value `e1`.

Contextual rules combine the context and the local transformation in one rule by using a local traversal that applies a rule that reuses information from the context. Indeed, in Stratego contextual rules are desugared to rules with a local traversal. Thus, the rule above corresponds to the non-contextual rule

```
InlineVar :
  |[ let var x := e1 in e2 end ]| ->
  |[ let var x := e1 in e2' end ]|
  where <oncetd((|[ x ]| -> |[ e1 ]|))> e2 => e2'
```

The problem with this approach is that it performs an extra traversal over the abstract syntax tree, leading to quadratic complexity in the case the contextual rule is applied as part of a traversal over the same tree that the context accesses.

```
DeclareFun =
  ?|[ function f(xs) : t = e ]|;
  rules(
    InlineFun :
      |[ f(es) ]| -> |[ let ds in e end ]|
      where <zip(BindVar)> (xs, es) => ds
  )
BindVar :
  (|[ x : t ]|, e) -> |[ var x : t = e ]|
```

Fig. 15. Dynamic definition of a function inlining rule.

8.3. Dynamic rules

In Visser (2001a) the extension of rewriting strategies with *scoped dynamic rewrite rules* was introduced. A dynamic rule is a normal rewrite rule that is defined at run-time and that can access information from its definition context. For example, to define an inliner, a rule that inlines function calls for a specific function can be defined at the point where the function is declared, and used at call sites of the function, as illustrated in Fig. 15. The `DeclareFun` strategy matches a function declaration and then *defines* a rule `InlineFun`, which inherits from its context the formal parameters `xs`, and the body of the function definition `e`. Thus, when applying the `InlineFun` rule to a call of the specific function `f` for which the rule was defined it is replaced with a let expression binding the actual parameters (`es`) to the formal parameters (`xs`) in the body of the function `e`.

Dynamic rules are first class. Their application is under control of a normal strategy. Thus dynamic rules can be applied as part of a global tree traversal. Rules can override the definition of previously defined rules. To restrict the application of a dynamic rule to a certain part of the tree, the live range of a rule can be determined by rule scopes. A rule temporarily overridden in a scope becomes visible again at the end of that scope. To hide rules defined in outer scopes, rules can be undefined. Rules from outer scopes can also be permanently overridden.

Dynamic rules turn out to be a very expressive extension of programmable rewriting strategies and has many applications. In Bravenboer and Visser (2002) it is shown how the combination of user-definable, generic traversals in combination with dynamic rules can be used to define the instruction selection strategies provided by BURG. In Olmos and Visser (2002) it is shown how dynamic rules can be used to define data-flow sensitive transformations on imperative programs. Other applications include interpretation (Dolstra and Visser, 2002), type checking, and partial evaluation (Olmos and Visser, 2003)

8.4. Regular path expressions

Another approach to context-sensitive rules is the use of *regular path expressions* as conditions in rewrite rules (Sittampalam et al., 2004). For example, the following rule expresses constant propagation:

```
ConstProp :
  |[ y := e[x] ]| -> |[ y := e[c] ]|
```

```
where fromentry({}*;
                {?|[ x := c ]|; <const> c};
                {not(def(x))}*;
                {use(x)})
```

by defining a rewrite on an assignment with an occurrence of a variable which is assigned a constant. This fact is expressed by the path expression in the condition of the rule, which states that there is a path from entry to the current node with an assignment assigning `c` to `x` and no redefinition of `x` in any node in between.

The applicability of such rewrite rules depends on an analysis of the entire procedure in which the assignment is embedded. In the approach described in Sittampalam et al. (2004) this is done automatically by the transformation system while traversing the tree. On every application of a rule the analysis needs to be recomputed. To make this feasible the analysis is performed incrementally, by maintaining for each node in the tree the partial matches to the regular expression. Thus, the reanalysis needs only to be performed on the path to the root of the tree. For this purpose the generic traversal strategies use the zipper data structure (Huet, 1997) for the tree representation to allow flexible navigation through the tree.

9. Discussion

9.1. Related work

Program transformation is a large research area with a long history. This survey gives an overview from the perspective of strategies in rule-based program transformation systems. In this overview many related aspects have been touched on. For each of these aspects more thorough surveys exist. Introductions to term rewriting in general include Dershowitz and Jouannaud (1990), Baader and Nipkow (1998) and Terese (2003). A survey of rewriting-based languages and systems is given in Heering and Klint (2003). The use of equations and rewriting for transformation is discussed in Field et al. (1998). There are special surveys for application areas of program transformation such as transformational programming (Feather, 1987; Partsch, 1990), reverse engineering (Chikofski and Cross, 1990; van den Brand et al., 1997), and application generation (Smaragdakis and Batory, 2000). Partsch and Steinbrüggen (1983) is a survey of early transformation systems. The 1999 Workshop on Software Transformation Systems (Sant'Anna et al., 1999) contains a series of articles reflecting on past experience with transformation systems. The Program Transformation Wiki (Visser et al., 2004) gives an overview of many types of program transformations, a list of transformation systems, and has elaborate special sections on *decompilation* and *reverse engineering* (van Deursen and Visser, 2002). Then there are areas that are not discussed in this survey, including graph transformation systems, abstract interpretation, reflective and generative approaches, and typing and correctness of transformation rules and strategies. Finally, this survey has concentrated on *mechanisms* for transformation not on specific transformations. An earlier version of this survey appeared as Visser (2001b).

9.2. Conclusion

Rule-based program transformation is going in the right direction. With recent developments in transformation languages more types of transformations can be expressed in rule-based formalisms. Recent additions such as dynamic rules and regular path queries drastically extend the expressiveness. Thus, an increasing number of transformation problems can be expressed concisely in a rule-based setting. Specification of control over rules while maintaining separation of rules and strategy is crucial. This does not mean that these solutions can always be used in production compilers, say, since dedicated implementations are still much faster. However, with the improvement of implementation techniques, but also just with the increase in computing power available, the size of problems that can be addressed by rule-based solutions increases. The main challenge for research in rule-based program transformation is the further expansion of the types of transformations that can be addressed in a natural way by accumulating the right abstractions.

Acknowledgements

I would like to thank Bernhard Gramlich and Salvador Lucas for inviting me to write the first version of this paper for the Workshop on Rewriting Strategies in 2001 and for their patience awaiting my delivery of this revised article. Jan Heering, Patricia Johann, Paul Klint, and Jurgen Vinju commented on a previous version of this paper. The comments by Oege de Moor and the anonymous referees for this Special Issue helped to improve the paper.

References

Aho, A., Sethi, R., Ullman, J., 1986. Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading, MA.

Appel, A.W., 1998. Modern Compiler Implementation in ML. Cambridge University Press.

Appel, A.W., Jim, T., 1997. Shrinking lambda expressions in linear time. Journal of Functional Programming 7 (5), 515–540.

Augusteijn, A., 1993. Functional Programming, Program Transformations and Compiler Construction. Ph.D. Thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands.

Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press.

Baeten, J., Bergstra, J., Klop, J., Weijland, W., 1989. Term-rewriting systems with rule priorities. Theoretical Computer Science 67 (2–3), 283–301.

Benzaken, V., Castagna, G., Frisch, A., 2003. CDuce: An XML-centric general-purpose language. In: Proceedings of the ACM International Conference on Functional Programming, pp. 51–64.

Bird, R.S., Meertens, L., 1987. Two exercises found in a book on algorithmics. In: Meertens, L. (Ed.), Program Specification and Transformation. North-Holland, pp. 451–458.

Borovanský, P., Cirstea, H., Dubois, H., Kirchner, C., Kirchner, H., Moreau, P.-E., Ringeissen, C., Vittek, M., 2000. ELAN V 3.4 User Manual, 4th edition. LORIA, Nancy, France.

Borovanský, P., Kirchner, C., Kirchner, H., 1996. Controlling rewriting by rewriting. In: Meseguer, J. (Ed.), Proceedings of the First International Workshop on Rewriting Logic and its Applications. WRLA'96. In: Electronic Notes in Theoretical Computer Science, vol. 4. Elsevier Science Publishers.

Borovansky, P., Kirchner, C., Kirchner, H., Moreau, P.-E., 2002. ELAN from a rewriting logic point of view. Theoretical Computer Science 285, 155–185.

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., Ringeissen, C., 1998. An overview of ELAN. In: Kirchner, C., Kirchner, H. (Eds.), Proceedings of the second International Workshop on Rewriting Logic and Applications. WRLA'98. In: Electronic Notes in Theoretical Computer Science, vol. 15. Elsevier Science Publishers.

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., Vittek, M., 1996. ELAN: a logical framework based on computational systems. In: Meseguer, J. (Ed.), Proceedings of the First Workshop on Rewriting Logic and Applications 1996. WRLA'96. In: Electronic Notes in Theoretical Computer Science, vol. 4. Elsevier Science Publishers.

Boyle, J.M., 1989. Abstract programming and program transformation—An approach to reusing programs. In: Biggerstaff, T.J., Perlis, A.J. (Eds.), Software Reusability, vol. 1. pp. 361–413.

Boyle, J.M., Harmer, T.J., Winter, V.L., 1997. The TAMPR program transformation system: Simplifying the development of numerical software. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (Eds.), Modern Software Tools for Scientific Computing. Birkhäuser, pp. 353–372.

Bravenboer, M., Visser, E., 2002. Rewriting strategies for instruction selection. In: Tison, S. (Ed.), Rewriting Techniques and Applications. RTA'02. In: Lecture Notes in Computer Science, vol. 2378. Springer-Verlag, pp. 237–251.

Bravenboer, M., Visser, E., 2004. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In: Schmidt, D.C. (Ed.), Proceedings of the 19th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. OOPSLA'04.

de Bruijn, N., 1980. A survey of the project AUTOMATH. In: To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalisms. Academic Press, pp. 579–606.

de Moor, O., Sittampalam, G., 2001. Higher-order matching for program transformation. Theoretical Computer Science 269 (1–2), 135–162.

de Moor, O., Backhouse, K., Swierstra, S.D., 2000. First class attribute grammars. In: Attribute Grammars and Their Applications. Informatica: An International Journal of Computing and Informatics 24 (2), 329–341.

Burstall, R.M., Darlington, J., 1977. A transformational system for developing recursive programs. Journal of the ACM 24 (1), 44–67.

Chikofski, E., Cross, J., 1990. Reverse engineering and design recovery: A taxonomy. IEEE Software 7 (1), 13–17.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F., 2002. Maude: specification and programming in rewriting logic. Theoretical Computer Science 285 (2), 187–243.

Collberg, C., Thomborson, C., Low, D., 1998. Manufacturing cheap, resilient and stealthy opaque constructs. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'98. pp. 184–196.

Cordy, J.R., Carmichael, I.H., Halliday, R., 1995. The TXL Programming Language, Version 8.

Cousot, P., Cousot, R., 2002. Systematic design of program transformation frameworks by abstract interpretation. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'02. pp. 178–190.

Czarnecki, K., Eisenecker, U.W., 2000. Intentional programming. In: Generative Programming. Methods, Tools, and Applications. Addison-Wesley (Chapter 11).

Dershowitz, N., Jouannaud, J.-P., 1990. Rewrite systems. In: van Leeuwen, J. (Ed.), Handbook of Theoretical Computer Science, vol. B. Elsevier Science Publishers, pp. 243–320 (Chapter 6).

Dijkstra, E.W., 1968. A constructive approach to the problem of program correctness. BIT 8 (3), 174–186.

Dolstra, E., Visser, E., 2002. Building interpreters with rewriting strategies. In: van den Brand, M.G.J., Lämmel, R. (Eds.), Workshop on Language Descriptions, Tools and Applications. LDTA'02. In: Electronic Notes in Theoretical Computer Science, vol. 65/3. Elsevier Science Publishers.

Elliott, C., Finne, S., de Moor, O., 2000. Compiling Embedded Languages. Springer-Verlag.

Feather, M.S., 1982. A system for assisting program transformation. ACM Transactions on Programming Languages and Systems 4 (1), 1–20.

Feather, M.S., 1987. A survey and classification of some program transformation approaches and techniques. In: Meertens, L.G.L.T. (Ed.), Program Specification and Transformation. In: IFIP, Elsevier Science Publishers, pp. 165–195.

Felty, A., 1992. A logic programming approach to implementing higher-order term rewriting. In: Eriksson, L.-H., Hallnäs, L., Schroeder-Heister, P. (Eds.), Extensions of Logic Programming. ELP'91. In: Lecture Notes in Artificial Intelligence, vol. 596. Springer-Verlag, pp. 135–158.

Field, J., Heering, J., Dinesh, T.B., 1998. Equations as a uniform framework for partial evaluation and abstract interpretation. ACM Computing Surveys 30 (3es), 2.

Fischer, B., Visser, E., 2004. Retrofitting the AutoBayes program synthesis system with concrete object syntax. In: Lengauer, C. et al. (Eds.), Domain-Specific Program Generation. In: Lecture Notes in Computer Science, vol. 3016. Spinger-Verlag, pp. 239–253.

Fitzpatrick, S., Stewart, A., Clint, M., Boyle, J.M., 1995. An algebra for deriving efficient implementations for an array processor parallel computer from functional specifications. Technical Report 1995/Jun-SF.AS.MC.JMB, Department of Computer Science, The Queen's University of Belfast, Northern Ireland.

Fokkink, W.J., Kamperman, J.F.T., Walters, H.R., 2000. Lazy rewriting on eager machinery. ACM Transactions on Programming Languages and Systems 22 (1), 45–86.

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

Fraser, C.W., Hanson, D.R., Proebsting, T.A., 1992a. Engineering a simple, efficient code-generator generator. ACM Letters on Programming Languages and Systems 1 (3), 213–226.

Fraser, C.W., Henry, R.R., Proebsting, T.A., 1992b. BURG—fast optimal instruction selection and tree parsing. ACM SIGPLAN Notices 27 (4), 68–76.

Futatsugi, K., Goguen, J., Jouannaud, J.-P., Meseguer, J., 1985. Principles of OBJ2. In: Reid, B. (Ed), Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'85. pp. 52–66.

Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B., 1977. Initial algebra semantics and continuous algebras. Journal of the ACM 24 (1), 68–95.

Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.-P., 2000. Introducing OBJ. In: Goguen, J.A., Malcolm, G. (Eds.), Software Engineering with OBJ: Algebraic Specification in Action. In: Advances in Formal Methods, Kluwer Academic Publishers, pp. 3–167 (Chapter 1).

Guttmann, W., Partsch, H., Schulte, W., Vullinghs, T., 2003. Tool Support for the Interactive Derivation of Formally Correct Functional Programs. Journal of Universal Computer Science 9 (2), 173–188.

Hatcher, W.S., Rus, T., 1976. Context-free algebras. Journal of Cybernetics 6, 65–76.

Heering, J., 1992. Implementing higher-order algebraic specifications. In: Miller, D. (Ed.), Proceedings of the Workshop on the λProlog Programming Language. University of Pennsylvania, Philadelphia, pp. 141–157. Also published as Technical Report MS-CIS-92-86 http://www.cwi.nl/~jan/HO.WLP.ps.

Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J., 1989. The syntax definition formalism SDF – reference manual. ACM SIGPLAN Notices 24 (11), 43–75.

Heering, J., Klint, P., 2003. Rewriting-based Languages and Systems, chapter 15, pp. 776–789. vol. 55 of Terese (2003).

Huet, G., 1997. Functional pearl: the zipper. Journal of Functional Programming 7 (5), 549–554.

Huet, G., Lang, B., 1978. Proving and applying program transformations expressed with second-order patterns. Acta Informatica 11, 31–55.

Johann, P., Visser, E., 2001. Fusing logic and control with local transformations: An example optimization. In: Gramlich, B., Lucas, S. (Eds.), Workshop on Reduction Strategies in Rewriting and Programming. WRS'01. In: Electronic Notes in Theoretical Computer Science, vol. 57. Elsevier Science Publishers.

Johnsson, T., 1987. Attribute grammars as a functional programming paradigm. In: Kahn, G. (Ed.), Functional Programming Languages and Computer Architecture. In: Lecture Notes in Computer Science, vol. 274. Springer-Verlag, pp. 154–173.

Jones, N., Gomard, C., Sestoft, P., 1993. Partial Evaluation and Automatic Program Generation. Prentice Hall.

Klop, J.W., 1992. Term rewriting systems. In: Abramsky, S., Gabbay, D., Maibaum, T. (Eds.), Handbook of Logic in Computer Science, vol. 2. Oxford University Press, Oxford, England, pp. 1–116.

Knuth, D.E., 1968. Semantics of context-free languages. Mathematical Systems Theory 2 (2), 127–145. Correction in: 1971. Mathematical Systems Theory, vol. 5(1). Springer-Verlag, pp. 95–96.

Kort, J., Lämmel, R., 2003. Parse-tree annotations meet re-engineering concerns. In: Third IEEE International Workshop on Source Code Analysis and Manipulation. SCAM'03. IEEE Computer Society Press., pp. 161–172.

Lacey, D., deMoor, O., 2001. Imperative program transformation by rewriting. In: Compiler Construction. CC'01. In: Lecture Notes in Computer Science, vol. 2027. Springer-Verlag, pp. 52–68.

Lämmel, R., 2003. Typed generic traversal with term rewriting strategies. Journal of Logic and Algebraic Programming 54, 1–64.

Lämmel, R., Peyton Jones, S.L., 2003. Scrap your boilerplate: a practical design pattern for generic programming. In: Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation. TLDI'03. ACM SIGPLAN Notices 38 (3), 26–37.

Lämmel, R., Riedewald, G., 2001. Prological language processing. In: van den Brand, M., Parigot, D. (Eds.), Proceedings of the First Workshop on Language Descriptions, Tools and Applications. LDTA'01. In: Electronic Notes in Theoretical Computer Science, vol. 44. Elsevier Science Publishers.

Lämmel, R., Visser, J., 2002. Typed combinators for generic traversal. In: Proceedings Practical Aspects of Declarative Programming PADL 2002. In: Lecture Notes in Computer Science, vol. 2257. Springer-Verlag, pp. 137–154.

Lämmel, R., Visser, J., Kort, J., 2000. Dealing with large bananas. In: Jeuring, J. (Ed.), Proceedings of the Workshop on Generic Programming. WGP'00. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Utrecht University, pp. 46–59..

Lämmel, R., Visser, E., Visser, J., 2002. The essence of strategic programming, October 2002 (Draft).

Lämmel, R., Visser, E., Visser, J., 2003. Strategic programming meets adaptive programming. In: Proceedings of Aspect-Oriented Software Development. AOSD'03. ACM Press, Boston, USA, March 2003, pp. 168–177.

Luttik, B., Visser, E., 1997. Specification of rewriting strategies. In: Sellink, M.P.A. (Ed.), 2nd International Workshop on the Theory and Practice of Algebraic Specifications. ASF+SDF'97. Springer-Verlag, Electronic Workshops in Computing.

Muchnick, S.S., 1997. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers.

Nadathur, G., Miller, D., 1988. An overview of λProlog. In: Kowalski, R.A. (Ed.), Logic Programming. Proceedings of the Fifth International Conference and Symposium, vol. 1. MIT Press, USA, pp. 810–827. 1988.

Neighbors, J.M., 1984. The Draco approach to constructing software from reusable components. IEEE Transactions on Software Engineering SE-10 (5), 564–573.

Ogata, K., Futatsugi, K., 1997. Implementation of term rewritings with the evaluation strategy. In: Proceedings 9th Symposium on Programming Languages: Implementations, Logics, and Programs. PLILP'97. In: Lecture Notes in Computer Science, vol. 1292. Springer-Verlag, pp. 225–239.

Olmos, K., Visser, E., 2002. Strategies for source-to-source constant propagation. In: Gramlich, B., Lucas, S. (Eds.), Workshop on Reduction Strategies. WRS'02. In: Electronic Notes in Theoretical Computer Science, vol. 70/6. Elsevier Science Publishers.

Olmos, K., Visser, E., 2003. Turning dynamic typing into static typing by program specialization. In: Binkley, D., Tonella, P. (Eds.), Third IEEE International Workshop on Source Code Analysis and Manipulation. SCAM'03. IEEE Computer Society Press, pp. 141–150.

Paige, R., 1994. Viewing a program transformation system at work. In: Hermenegildo, M., Penjam, J. (Eds.), Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming. In: Lecture Notes in Computer Science, vol. 844. Springer-Verlag, pp. 5–24.

Paige, R., 1996. Future directions in program transformations. Computing Surveys 28A (4).

Parr, T.J. et al., 2003. ANTLR reference manual. http://www.antlr.org, January 2003. ANTLR Version 2.7.2.

Partsch, H., 1986. Transformational program development in a particular problem domain. Science of Computer Programming 7 (2), 99–241.

Partsch, H., 1990. Specification and Transformation of Programs. A Formal Approach to Software Development. Springer-Verlag.

Partsch, H., Schulte, W., Vullinghs, T., 1999. System support for the interactive transformation of functional programs. In: Proceedings of the 21st International Conference on Software engineering (Sant'Anna et al., 1999). pp. 701–702. The actual proceedings are available at http://www.dur.ac.uk/CSM/STS/.

Partsch, H., Steinbrüggen, R., 1983. Program transformation systems. ACM Computing Surveys 15 (3), 199–236.

Pettorossi, A., Proietti, M., 1996a. Future directions in program transformation. ACM Computing Surveys 28 (4es), 171–es. December 1996. Position Statement at the Workshop on Strategic Directions in Computing Research. MIT, Cambridge, MA, USA, June 14–15.

Pettorossi, A., Proietti, M., 1996b. Rules and strategies for transforming functional and logic programs. ACM Computing Surveys 28 (2), 360–414.

Peyton Jones, S.L., Marlow, S., 2002. Secrets of the Glasgow Haskell Compiler inliner. Journal of Functional Programming 12 (4), 393–434.

Peyton Jones, S.L., Santos, A.L.M., 1998. A transformation-based optimiser for Haskell. Science of Computer Programming 32 (1–3), 3–47.

Peyton Jones, S.L., Tolmach, A., Hoare, T., 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In: Hinze, R. (Ed.), 2001 Haskell Workshop. ACM SIGPLAN.

Pfenning, F., Elliot, C., 1988. Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'88. ACM, pp. 199–208.

Pitts, A.M., Gabbay, M.J., 2000. A metalanguage for programming with bound names modulo renaming. In: Backhouse, R., Oliveira, J.N. (Eds.), Proceedings of the 5th International Conference on Mathematics of Programme Construction. MPC2000. In: Lecture Notes in Computer Science, vol. 1837. Springer-Verlag, pp. 230–255.

Proebsting, T.A., 1995. BURS automata generation. ACM Transactions on Programming Languages and Systems 17 (3), 461–486.

Reps, T., Teitelbaum, T., 1988. The Synthesizer Generator: A System for Constructing Language-Based Editors. Springer-Verlag.

Rich, C., Waters, R.C., 1990. The Programmer's Apprentice. In: Frontier Series, ACM Press.

Roberts, D., Brant, J., Johnson, R.E., 1997. A refactoring tool for Smalltalk. Theory and Practice of Object Systems 3 (4), 253–263.

Sant'Anna, M., Leite, J., Baxter, I., Wile, D., Biggerstaff, T., Batory, D., Devanbu, P., Burd, L., 1999. International workshop on software transformation systems (STS'99). In: Proceedings of the 21st International Conference on Software Engineering. IEEE Computer Society Press, pp. 701–702. The actual proceedings are available at http://www.dur.ac.uk/CSM/STS/.

Saraiva, J., Kuiper, M., 1998. LRC—A generator for incremental language-oriented tools. In: Koskimies, K. (Ed.), 7th International Conference on Compiler Construction. CC'98. In: Lecture Notes in Computer Science, vol. 1383. Springer-Verlag.

Saraiva, J., Swierstra, S.D., 1999. Data structure free compilation. In: 8th International Conference on Compiler Construction. CC'99. In: Lecture Notes in Computer Science, vol. 1575. Springer-Verlag, pp. 1–16.

Sheard, T., Peyton Jones, S.L., 2002. Template metaprogramming for Haskell. In: Chakravarty, M.M.T. (Ed.), ACM SIGPLAN Haskell Workshop 02. pp. 1–16.

Sittampalam, G., de Moor, O., Larsen, K.F., 2004. Incremental execution of transformation specifications. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'04, pp. 26–38.

Smaragdakis, Y., Batory, D., 2000. Application generators. In: Webster, J. (Ed.), Encyclopedia of Electrical and Electronics Engineering. John Wiley and Sons.

Smith, D.R., 1990. KIDS: A semiautomatic program development system. IEEE Transactions on Software Engineering 16 (9), 1024–1043.

Taha, W., Sheard, T., 2000. MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science 248 (1–2), 211–242.

Terese, 2003. Term Rewriting Systems. In: Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press.

van den Brand, M.G.J., Visser, E., 1996. Generation of formatters for context-free languages. ACM Transactions on Software Engineering and Methodology 5 (1), 1–41.

van den Brand, M.G.J., Klint, P., Verhoef, C., 1997. Reverse engineering and system renovation: an annotated bibliography. ACM Software Engineering Notes 22 (1), 42–57.

van den Brand, M.G.J., Klint, P., Vinju, J.J., 2003. Term rewriting with traversal functions. ACM Transactions on Software Engineering and Methodology 12 (2), 152–190.

van den Brand, M.G.J., Sellink, M.P.A., Verhoef, C., 2000b. Generation of components for software renovation factories from context-free grammars. Science of Computer Programming 36, 209–266.

van den Brand, M.G.J., de Jong, H., Klint, P., Olivier, P., 2000a. Efficient annotated terms. Software, Practice & Experience 30 (3), 259–291.

van Deursen, A., Visser, E., 2002. The reengineering Wiki. In: Proceedings 6th European Conference on Software Maintenance and Reengineering. CSMR'02. IEEE Computer Society, pp. 217–220.

van Deursen, A., Klint, P., Tip, F., 1993. Origin tracking. Journal of Symbolic Computation 15 (5–6), 523–546.

van Deursen, A., Heering, J., Klint, P. (Eds.), 1996. Language Prototyping. An Algebraic Specification Approach. AMAST Series in Computing, vol. 5. World Scientific, Singapore.

van de Pol, J., 2001. Just-in-time: On strategy annotations. In: International Workshop on Reduction Strategies in Rewriting and Programming. WRS'01. In: Electronic Notes in Theoretical Computer Science, vol. 57.

Visser, E., 1999. Strategic pattern matching. In: Narendran, P., Rusinowitch, M. (Eds.), Rewriting Techniques and Applications. RTA'99. In: Lecture Notes in Computer Science, vol. 1631. Springer-Verlag, pp. 30–44.

Visser, E., 2000. Language independent traversals for program transformation. In: Jeuring, J. (Ed.), Workshop on Generic Programming. WGP'00. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Utrecht University.

Visser, E., 2001a. Scoped dynamic rewrite rules. In: van den Brand, M., Verma, R. (Eds.), Rule Based Programming. RULE'01. In: Electronic Notes in Theoretical Computer Science, vol. 59/4. Elsevier Science Publishers.

Visser, E., 2001b. A survey of rewriting strategies in program transformation systems. In: Gramlich, B., Lucas, S. (Eds.), Workshop on Reduction Strategies in Rewriting and Programming. WRS'01. In: Electronic Notes in Theoretical Computer Science, vol. 57. Elsevier Science Publishers.

Visser, E., 2002. Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (Eds.), Generative Programming and Component Engineering. GPCE'02. In: Lecture Notes in Computer Science, vol. 2487. Springer-Verlag, pp. 299–315.

Visser, E., 2004. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In: Lengauer, C. et al. (Eds.), Domain-Specific Program Generation. In: Lecture Notes in Computer Science, vol. 3016. Spinger-Verlag.

Visser, J., 2001. Visitor combination and traversal control. In: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications. OOPSLA'01. pp. 270–282.

Visser, E., Benaissa, Z.-e.-A., Tolmach, A., 1998 Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming. ICFP'98. pp. 13–26.

Visser, E. et al., 2004. The Program Transformation Wiki. http://www.program-transformation.org.

Vogt, H., 1989. Higher-order attribute grammars. Ph.D. Thesis, Department of Computer Science, Utrecht University.

Wadler, P., 1987. Views: A way for pattern matching to cohabit with data abstraction. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'87. ACM, pp. 307–313.

Waters, R.C., 1982. The programmer's apprentice: Knowledge based program editing. IEEE Transactions on Software Engineering 8 (1), 1–12.