A Survey of Strategies in Program Transformation Systems

Eelco Visser

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

Abstract. Program transformation is used in a wide range of applications including compiler construction, optimization, program synthesis, refactoring, software renovation, and reverse engineering. Complex program transformations are achieved through a number of consecutive modifications of a program. Transformation rules define basic modifications. A transformation strategy is an algorithm for choosing a path in the rewrite relation induced by a set of rules. This paper surveys the use of strategies in program transformation systems. After a discussion of kinds of program transformation and choices in program representation, the basic elements of a strategy system are discussed and the choices in the design of a strategy language are considered. Several styles of strategy systems as provided in existing languages are then analyzed.

1 Introduction

Program transformation has applications in many areas of software engineering such as compilation, optimization, refactoring, program synthesis, software renovation, and reverse engineering. 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.

Many systems for program transformation exist that are often specialized for a specific object language and/or kind of transformation. All these systems share many ideas about program transformation and use similar techniques, but are often ad-hoc in many respects. The ultimate goal is to achieve a specification language or family of specification languages for the high-level, declarative specification of program transformation systems in which generic, language independent schemas of transformation can be captured, and which admits efficient implementation of those transformations that can scale up to be applied to large programs.

This survey aims at understanding the similarities and differences between systems for program transformation by analysing existing systems based on publications. Many aspects of program transformation such as parsing, pretty-printing and formulating basic transformations are fairly well understood. Therefore, this survey concentrates on transformation strategies, i.e., the control part of a transformation system that determines the order of application of basic transformation steps.

The paper is structured as follows. Section 2 presents a taxonomy of program transformation. Section 3 discusses the choices in program representations that can be used in program transformation systems. Section 4 dicusses the ingredients of a system for the specification of program transformation rules and strategies. Section 5 discusses the implementation of program transformation in a number of dedicated transformation languages. Section 6 summarizes and discusses some areas for research in the implementation of transformation systems

2 Program Transformation

A program is a structured object with semantics. The structure allows us to transform a program. Semantics includes the extensional and intensional behaviour of a program and gives us the means to compare programs and to reason about the validity of transformations. A programming language is a collection of programs that comply with the same set of structural and semantic rules. This is a rather broad definition that is intended to cover proper programming languages (with an operational interpretation), but also data formats, domain-specific languages, and aspects of programs such as their control- or data-flow.

Programming languages can be clustered into classes with structural and/or semantic similarities. One of the aims of a general framework for program transformation is to define transformations that are reusable accross as wide a range of languages as possible. For example, the notion of variable and variable binding is shared by all programming languages. Transformations dealing with variables such as bound variable renaming, substitution, and unification can be defined in a very generic manner for all languages at once.

Program transformation is the act of changing one program into another. The term *program transformation* is also used for a formal description of an algorithm that implements program transformation. The language in which the program being transformed and the resulting program are written are called the *source* and *target* languages, respectively.

Program transformation is used in many areas of software engineering, including compiler construction, software visualization, documentation generation, and automatic software renovation. In all these applications we can distinguish two main scenarios, i.e., ones where the source and target language are different (*translations*) from scenarios where they are the same (*rephrasings*). These main scenarios can be refined into a number of typical sub-scenarios based on their effect on the level of abstraction of a program and to what extent they preserve the semantics of a program. This refinement results in the taxonomy in Figure 1.

2.1 Translation

In a *translation* scenario a program is transformed from a source language into a program in a *different* target language. Translation scenarios can be distinguished by their effect on the level of abstraction of a program. Although translations aim at preserving the extensional semantics of a program, it is usually not possible to retain all information across a translation. Examples of translation scenarios are synthesis, migration, reverse engineering, and analysis. Their relations are illustrated by the diagram in Figure 2.

Synthesis Program *synthesis* is a class of transformations that lower the level of abstraction of a program. In the course of synthesis design information is lost. In *refinement* [44] an implementation is derived from a high-level specification such that the implementation satisfies the specification. *Compilation* [1, 2, 32] is a form of synthesis in which a program in a high-level language is transformed to machine code. 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. Other examples of synthesis are parser and pretty-printer generation from context-free grammars [1, 10].

Migration 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.

Reverse Engineering The purpose of *reverse engineering* [11] 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.

Translation

- Migration
- Synthesis
  - Refinement
  - Compilation
- Reverse engineering
  - Decompilation
  - Architecture extraction
  - Documentation generation
  - Software visualization
- Analysis
  - Control-flow analysis
  - Data-flow analysis

Rephrasing

- Normalization
  - Simplification
  - Desugaring
  - Weaving
- Optimization
  - Specialization
  - Inlining
  - Fusion
- Refactoring
  - Design improvement
  - Obfuscation
- Renovation

Fig. 1. A taxonomy of program transformation

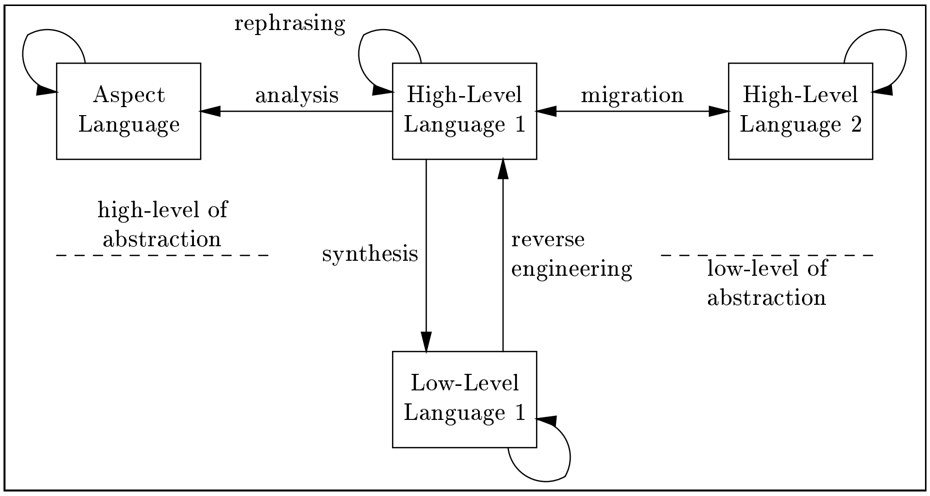

Fig. 2. Relation between kinds of program transformation

Analysis Program *analysis* reduces a program to one aspect such as its control- or data-flow. Analysis can thus be considered a transformation to a sub-language or an aspect language.

2.2 Rephrasing

Rephrasings are transformations that transform a program into a different program in the *same* language, i.e., source and target language are the same. In general, rephrasings try to say the same thing in different words thereby aiming at *improving* some aspect of the program, which entails that they *change the semantics* of the program. The main sub-scenarios of rephrasing are normalization, optimization, refactoring, and renovation.

Normalization A *normalization* reduces a program to a program in a sub-language, with the purpose of decreasing its syntactic complexity. *Desugaring* is a kind of normalization in which some of the constructs (syntactic sugar) of a language are eliminated by translating them into more fundamental contructs. For example, the Haskell language definition [40] describes for many constructs how they can be desugared to a kernel language. Other examples are module flattening and the definition of EBNF constructs in pure BNF as is done for example in the SDF2 normalizer [48]. *Simplification* is a more general kind of normalization in which a program is reduced to a normal (standard) form, without necessarily removing simplified constructs. For example, consider canonicalization of intermediate representation and algebraic simplification of expressions. Note that normal form does not necessarily correspond to being a normal form with respect to a set of rewrite rules.

Optimization An *optimization* [2, 32] is a transformation that improves the run-time and/or space performance of a program. Examples of optimization are fusion, inlining, constant propagation, constant folding, common-subexpression elimination, and dead code elimination.

Refactoring A *refactoring* [22] is a transformation that improves the design of a program by restructuring it such that it becomes easier to understand while preserving its functionality. *Obfuscation* [15] 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.

Renovation In *software renovation* the extensional behaviour 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.

2.3 Program Transformation Systems

A program transformation system is determined by the choices it makes in program representation and the programming paradigm used for implementing transformations. The next section discusses considerations in choosing a representation for programs. The remaining sections consider implementation of transformations.

3 Program Representation

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. Since programs are written as texts by programmers, parsers are needed to convert from text to structure and unparsers are needed to convert structure to text. However, this might change in the future; in the Intentional Programming framework [16] programs are stored, edited and processed as source graphs. Editing of programs is done via structure editors.

A number of issues should be considered when choosing a program representation: to use parse trees or abstract syntax trees, trees or graphs, how to represent variables and variable bindings, and how to exchange programs between transformation components.

Parse Trees or Abstract Syntax Trees Parse trees contain syntactic information such as layout (whitespace 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. Especially the latter aspect is problematic; it is not clear in a generic manner where to insert comments in a transformed fragment of a program. Origin tracking [47] might be useful here.

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.

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 in 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 sub-trees are shared by multiple contexts. The advantage of sharing is reduced memory usage. In the ATerm library [45] this approach is taken to the extreme by only constructing one instance for each sub-tree that is constructed, thus achieving maximal sharing and minimal memory usage.

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 re-building the context in which the transformed tree is used. It would be more attractive to overwrite the root node of the sub-tree 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 occurences 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 occurences.

The same problem of occurence appears when associating information with nodes. When sharing is based on syntactic equivalence alone, annotations become associated with all occurences 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.

Finally, full fledged graphs can be useful to represent back-links in the tree to represent, for example, loops in a control-flow graph [2, 30, 32], or links to declarations [16]. Updatable 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 sub-graph 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.

Variable Bindings 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 occurences out of bindings.

Transparent handling of variable bindings is desirable. Higher-order abstract syntax (HOAS) [17, 27, 41] 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 [42] 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 structured described by the grammar for more complex binding schemes.

All approaches that rename variables are in conflict with requirements that original names are preserved, which is required in applications such as refactoring and renovation.

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 occurences of a symbol with the information in the declarations. Either way, the information needs to be kept consistent during transformations.

Exchange Format Finally, a program representation should be supported by an exchange format that makes it possible to exchange programs between transformation components. Example formats are XML, which supports exchange of tree shaped data, and the ATerm Format [45], which supports exchange of DAGs, maintaining maximal sharing. See [26] for a bibliography of exchange formats.

4 Implementation of Program Transformation

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.

For example, consider the transformation rules in Figure 3. 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 get rid of the constant variable definitions:

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

On the other hand, the ExtractFunction 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)
```

Rules can be applied interactively by a programmer via a graphical user interface. The problem with such manipulations is that the transformation is not reproducible, since the decisions have not been recorded. By automating the transformation process, a series of basic transformations can be repeatedly applied to a program. By generalizing the sequence of transformations the combined transformation can be applied to many programs. This requires a mechanism for combining rules into more complex transformations. This section discusses the basic ingredients for specification of rules and strategies.

4.1 Transformation Rules

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 behaviour 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.

A rule consists of recognizing a program fragment to transformation and constructing a new program fragment to replace the old one. Recognition involves

```
rules
  InlineF : let f(xs) = e in e'[f(es)] -> let f(xs) = e in e'[e[es/xs]]
  InlineV : let x = e in e'[x] -> let x = e in e'[e]
  Dead : let x = e in e' -> e' where <not(in)> (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 <not(in)> (x, <free-vars> e2)
```

Fig. 3. Some example transformation rules

matching the syntactic 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 that performs arbitrary side-effects.

When using a tree or term representation *term pattern matching* can be used. First-order term patterns can be 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 leafs, 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 [19] is used to divide a list in multiple sublists possibly separated by element patterns. *Associative-commutative (AC) matching* in OBJ, Maude [12] and ELAN [4] supports the treatment of lists as multi-sets. *Higher-order matching* in λProlog [33, 41] allows the matching of subterms at a variable depth. *Views* for Haskell, as proposed in [54], provide a way to view a data structure using different patterns than are used to represent them. *Overlays* in Stratego [49] are pseudo-constructors that abstract from an underlying representation using real constructors.

4.2 Transformation Strategies

A set of transformation rules for a programming language induces a rewrite relation on programs [18]. 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 Figure 4, 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 commutivity rules), and non-confluence in which a program can be transformed into two different programs.

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. 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.

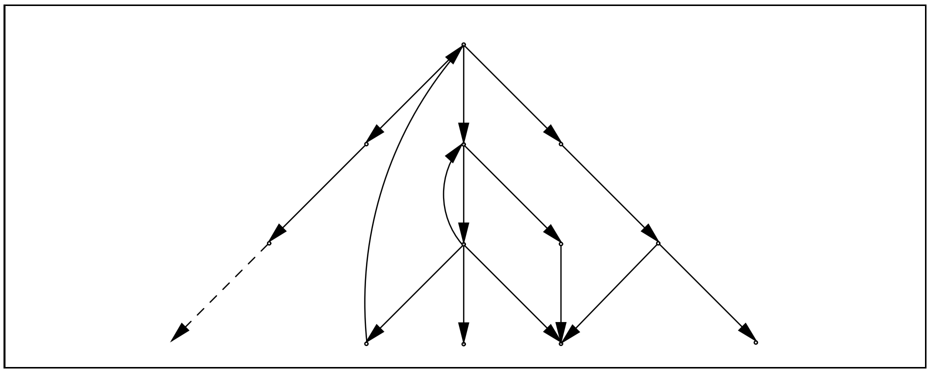

Fig. 4. Phenomena in composition of transformation rules: infinite branches, inverses, confluence, non-confluence

A strategy can be provided by the transformation engine or can be user-definable. There is a range of possibilities between these two options:

- 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 or lazyness 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 lazyness information.
- Completely programmable. The strategy to apply rules can be programmed in a strategy language.

Whether defined by the user or by the engine, the strategy needs to be expressed and implemented formally. The rest of this section considers the ingredients of a language for defining strategies.

Sequential Composition To choose a path in the rewrite relation the basic rules should be combined into transformations. Transformations can be combined by consecutively applying two transformations, by conditionally choosing between two transformations, and by repeating transformations, iteratively or recursively.

Non-Deterministic Programming Choosing between two paths based on properties of the current program can be too limited. It might be necessary to decide on the appropriateness of a choice after applying several transformations. One approach is to speculatively explore paths until an acceptable solution is found, the other is to explore all paths in parallel and choose the best solution.

In the case of speculative exploration some kind of non-deterministic choice between two alternative paths is needed. On failure in one of the paths, the other path is taken. If backtracking is local the choice is committed after one of the chosen branches succeeds. If back-tracking is global, failure at any point after the choice causes back-tracking. This allows exploring the full search space, until an acceptable solution is found.

Parallel exploration of all paths requires a mechanism for comparing solutions based on some kind of cost function. For some such problems dynamic programming techniques can be used to efficiently apply all transformations.

In between speculative and parallel exploration is goal based exploration in which a set of constraints leads to discarding paths with inconsistent constraints.

Structure Traversal A rewrite relation includes application of rules in any context. This entails traversing the program representation structure to find the location where the rule is applied, applying the rule, and rebuilding the context. In addition to the sequential order of application and the choice between paths, the location where a rule is applied is also determined by the strategy. It is unnatractive to express these locations by means of paths in a tree, since it is inneficient to traverse and re-build the context for each rule application. Often rules are applied close to each other in the tree.

Therefore, some mechanism is needed to traverse syntax trees and to apply transformation rules at sub-trees. A language specific traversal mechanism requires definition of traversal for construct in a language. This can lead to large specifications for large languages (having of a complex abstract syntax). A language generic traversal mechanism supports the definition of generic traversals over abstract syntax trees. This requires exposing the underlying representation model. Traversal mechanisms can provide a set of fixed traversals such as top-down and bottom-up, or provide traversal primitives from which all kinds of traversals can be defined.

Information Carrying Strategies Strategies may carry information that can be used in making decisions about paths to take and to pass context-sensitive information to rules.

Separation of Rules and Strategies Although we have distinghuished rules at the conceptual level, at the implementation level rules and strategies can be intertwined, i.e., the rules are hardwired in the definition of the strategy. Alternatively rules and strategies are defined separately, which entails that strategies are parameterized with a set of rules.

Separate definition of rules and strategies leads to clearer specifications that allow reasoning about smaller entitities (rules, strategies) separately. Furthermore, separated definition enables the reuse of rules and strategies and the generic implementation of aspects of transformation systems that are common to all or a class of languages. However, intertwining may sometimes be required for efficiency reasons. In these cases it desirable that the intertwining be done by a compiler rather than by the specifier.

Abstraction To achieve reuse of strategies in general, and separation of rules from strategies, in particular, an abstraction mechanism is needed that allows abstraction over rules and strategies. That is, it should be possible to name and parameterize rules and strategies.

5 Program Transformation Languages

This section discusses a number of languages designed specifically for the implementation of program transformation. The following topics will be discussed: interactive program transformation, attribute grammars, tree parsing, term rewriting and a number of extensions of basic term rewriting addressing the problems of standard rewriting strategies: strategy annotations, sequential normalization, exploring the reduction space with non-determinstic strategies, guiding rewriting by reflection, rewriting with traversals, and generic rewriting strategies.

5.1 Interactive Program Transformation

Draco [35, 36] was the first system to support the transformation of high-level domain-specific programs to executable code. The system supported the definition of transformation rules for optimizations and refinements to transform high-level constructs into lower-level ones.

Transformation rules and refinements are identified by means of names. Transformation rules also have an application code that specifies their relative precedence. The application of transformations and refinements to a domain program is controlled by the user through an interactive process. In this process the user has to select domain, instance (region in the abstract syntax tree representing the program), and locale (node in the abstract syntax tree). Transformations can be applied directly to the currently selected locale using APPLY. The system can examine the tree and SUGGEST transformations to apply. Using the TRANSFORM command all transformation rules in a certain range can be applied automatically. The TRANSFORM uses a bottom-up traversal over the tree, applying rules in the provided code range. Rules with higher codes are applied first. In [34] there are also descriptions of a top-down traversal and of traversals that apply the best rules first. Refinements can be applied individually using TRY and USE. A certain amount of automation of the process is possible by means of tactics.

5.2 Intentional Programming

Intentional Programming is a meta-programming system under development at Microsoft Research by Charles Simonyi et al. A good description of Intentional Programming is given in [16].

In Intentional Programming a program is represented by a source tree instead of by a source text. Each node of a source tree has a reference to its declaration (thus making the tree into a source graph). For example, an occurence of a variable has (is) a link to its declaration. Likewise each language construct, or intention, corresponds to a tree node that defines it. Intentions can be used by making a link to the definition of the intention. For example, a while statement is a node with two children corresponding to the condition and the iteration statement and a link to the `while` intention. Domain-specific programming abstractions can be captured by defining new intentions.

Source trees are implemented by reducing them to a source tree using only intentions R-codes intentions. R-code intentions are basic constructs that can translated to some form of machine code by a code generator. Part of the definition of each intention is a method reducing it to its R-code. The dependencies between these reduction methods are computed and interpreted by the IP engine.

5.3 Simple 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. SORCERER [38, 39] is the tree parser generator for the ANTLR language processing system. SORCERER 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 SORCERER.

5.4 Tree Parsing with Dynamic Programming

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 to 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 [23, 24, 43] is a system for code generation from intermediate expression trees. A mapping from IR trees to machine instruction 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 [43] gives the example grammar in Figure 5. 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)))`.

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 righ-hand sides of the productions to the tree, taking into account earlier matches for sub-trees. 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 [43] for simple type inference problems, for checking tree formats, and for tree simplifications.

5.5 Term Rewriting

Term rewriting [18] is supported by many systems including OBJ [25], ASF+SDF [19], and ELAN [5]. Term rewriting is an attractive paradigm for program transformation. First-order terms can be used to describe the abstract syntax of programs.

```
[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. 5. Example BURG specification

For example, consider the declaration of propositional formulae in Figure 6. A rewrite rule of the form `t1 -> t2` declares the transformation of a term matching pattern `t1` to the instantiation of `t2`. Rewrite rules can be used to express basic transformation rules and can be considered as operationalizations of algebraic laws of the language. For example, the rewrite rules in Figure 6 express basic laws of propositional logic, i.e., distribution rules, the rule of double negation, and the De Morgan rules. Using stronger forms of pattern matching such as various instances of equational matching (e.g., AC matching, list matching), patterns can capture complicated term configurations. Furthermore, in conditional rewrite rules additional tests on the pattern can be stated.

A redex is a sub-term that matches with a rewrite rule. A term is in normal form if it has redexes. Rewrite engines for term rewrite systems compute for a given term its normal form with respect to the set of rules in a specification. This involves exhaustively applying rules to sub-terms until no more rules apply. A rewrite engine can employ different strategies to order the application of rules. In innermost rewriting all sub-terms of a term are normalized before rules are applied to the term itself. In outermost rewriting redexes closest to the root of the term are rewritten first. This implies that rules are automatically applied throughout a term and that no traversals over the syntax tree need to be defined.

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 Figure 6. This rewrite system is non-terminating because rules `DAOL` and `DAOR` enable rules `DOAL` and

```
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. 6. Rewrite system for propositional formulae.

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.

The common solution to this kind of problem is to introduce additional constructors (functions) that achieve normalization under a restricted set of rules. Figure 7 shows how the rewrite system in Figure 6 can be turned into a terminating rewrite system that defines the normalization to disjunctive-normal form. 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 Prop 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* [20]. These rules assume that their arguments have already been normalized to disjunctive normal form. 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

```
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. 7. Functionalized rewrite system for disjunctive normal form.

only apply if the other rules do not apply. If this mechanism had not been used even more rules had to be used to handle the cases were the real transformation rules do not apply. Default rules were introduced in ASF+SDF [19].

The kind of problem illustrated in the example above occurs frequently in all kinds of transformations. Examples are the normalization of SDF2 syntax definitions to Kernel-Sdf [48]; desugaring of programming constructs; refactoring in which parts of a program may have to simplified, while others may have to de-simplified.

In general, trying to overcome the problems of non-termination and non-confluence leads to explicit specification of control in rewrite rules by defining new constructors (functions). This usually leads to a functional programming style of rewriting, overhead in the form of traversal rules for each constructor in the signature, intertwining of rules in function definitions, make reuse of rules impossible, and specifications that are much harder to understand.

5.6 Rewriting with Traversal Functions

In ASF+SDF controlling the application of transformation rules has been recognized as a problem for a long time. Especially for the specification of transformations for large languages such as COBOL the overhead of defining traversals was seen as the problematic factor. First this was solved by the generation of default traversal rules [10, 9] that can be overridden by normal rules. In this approach typically only a few rewrite rules have to be specified, corresponding with the non-default behaviour of the traversal. However, the number of generated rules still proves to be a source of overhead, be it for the compiler, not the programmer. Furthermore, providing a new traversal scheme required the addition of a new generator.

In a recent approach [8], traversal functions are supported directly by the rewriting engine, avoiding the compile-time overhead of generated rules. Figure 8 illustrates the approach applied to the problem of normalization to disjunctive normal form. Note that the example does not use ASF+SDF syntax. The specification is the same as that in Figure 7, but the `dnf` function has been declared a traversal function in the signature. The attribute `traversal(bottom-up)` 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 sub-terms) before it is applied at the term itself. Only for those constructors that need to be transformed, a rule needs to be declared; the default behaviour is to reconstruct the term with the original constructor. In the example this reduces the specification of the traversal from 6 to 2 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 Figure 8 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 Figure 9 in which no smart constructors are used. In this style only one rule is needed for each original rule. However, the problem with this style is that the

```
signature
  constructors
    dnf : Prop -> Prop {traversal(bottom-up)}
    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. 8. Disjunctive Normal Form with traversal function (Version 1)

recursive calls in the right-hand sides of the rules will completely re-traverse the tree (the arguments of which are already normalized) before applying one of the rules.

ASF+SDF provides a limited set of traversals. For traversal strategy there is a choice between `top-down` and `bottom-up`. The latter has been explained above. A `top-down` traverses down the tree and stops as soon as a rule applies. In addition a traversal can be a transformation (`trafo`) and/or a traversal wich accumulates information along the way (`accu`). Finally, traversal functions can be parameterized with additional arguments that contain static information to be used during traversal.

The advantage of traversal functions is that default traversal behaviour does not need to be implemented manually. This is similar to default visitors in object-

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

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

Fig. 9. Disjunctive Normal Form with traversal function (Version 2)

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, and to argue about correctness of the whole.

Secondly, the traversal function schema provides a limited ranges of traversals. The bottom-up variant does a full traversal of the tree. The top-down variant stops as soon as it has found a rule application, this requires explicit defininition of recursion in rules. Although the authors claim [8] that using these it is possible to implement all kinds of traversals, this requires gluing together the basic traversals in an ad-hoc manner. It is not possible to define new traversal schemas in a reusable way, i.e., as a new traversal attribute. That would require extending the rewrite engine.

Finally, the traversals provided by the language capture an abstraction, i.e., certain traversal schemata. There is no possibility in the language to give further abstractions. Either alternative traversal schemata, or more elaborate functionality involving traversals. This 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.

5.7 Term Rewriting with Strategy Annotations

Another 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 Figure 10 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. Using an outermost strategy would solve the problem, but outermost is harder to implement efficiently. Therefore, several systems provide *strategy annotations* to delay the evaluation of arguments.

Note that these strategy annotations help to make some rewrite systems terminating, but that do not help in other respects for program transformation. For example, traversals over abstract syntax trees still need to be defined explicitly.

Just-in-time The strategy annotations in [46] 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 redexes.

A strategy annotation for a constructor is a list of argument positions and rule names. The argument positions indicate the next argument to evaluate,

```
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
  IfE : If(p, x, x) -> x
```

Fig. 10. Rewrite system with non-terminating reduction path.

the rule name 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 Figure 10 is solved by means of the annotation

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

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

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 Figure 11. Terms of the form `Inf(n)`, for some natural number `n`, do not have a normal form.

The E-strategy [37] of the CafeOBJ system uses an extended form of strategy annotations in which not al 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.

For example, the non-termination in Figure 11 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)))`.

Lazyness annotations The strategy annotations discussed above are interpreted by the rewrite engine. In [21] it is shown how rewrite systems with lazy-

```
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. 11. Specification with strategy annotations [37].

ness annotations can be compiled into rewrite systems that can be evaluated using an innermost strategy.

A lazyness annotation indicates for an argument of a constructor that it is lazy, i.e., that no reductions should be performed for sub-terms of that argument, unless needed for matching. For example, for the rewrite system in Figure 11 the lazyness annotation `Lazy(Cons,2)` would achieve the delay of the evaluation of the second argument of `Cons`.

A rewrite system with lazyness 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 TRS in Figure 11 is transformed to the eager TRS in Figure 12. 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.

5.8 Sequence of Canonical Forms

TAMPR stands for *Transformation Assisted Multiple Program Realisation System*. The TAMPR system [6, 7], which has been in use since the seventies, is designed for the derivation of efficient implementations from a specification through transformations, in particular in the domain of numerical programming.

```
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. 12. Result of translating specification with lazyness annotations to eager specification [21].

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 that we encountered in Section 5.5 is solved in TAMPR by organizing a large transformation into a sequence of consecutive canonicalizations under different sets of rewrite rules. Typically such a sequence starts with several preparatory steps that bring the program in the right form, followed by the pivotal step which achieves the actual transformation, followed by some post-processing.

In [7] this is illustrated with the transformation from a polynomial in y:

$$(x^2 + 2x + 1)y^2 + (x^2 - 9)y - (20x^2 + 18x - 18)$$

to the equivalent polynomial in x

$$(y^2 + y - 20)x^2 + (2y^2 - 18)x + (y^2 - 9y + 18)$$

This is achieved by means of the following pipeline of canonicalizations:

```
sum-of-monomonials;
x-commuted-to-right;
like-powers-collected;
x-factored-out
```

The `sum-of-monomonials` transforms the polynomial into

$$x^2y^2 + 2xy^2 + y^2 + x^2y - 9y - 20x^2 - 18x + 18$$

By commuting the multiplications, the `x-commuted-to-right` canonical form is achieved:

$$y^2x^2 + 2y^2x + y^2 + yx^2 - 9y - 20x^2 - 18x + 18$$

The `like-powers-collected` canonical form commutes the additions to bring monomonials with the same power of x together:

$$y^2x^2 + yx^2 - 20x^2 + 2y^2x - 18x + y^2 - 9y + 18$$

Finally, by factoring out the powers of x, the desired form is reached.

5.9 Non-deterministic Sequential Strategies

ELAN [3] is a language for rewriting with user-definable strategies in a special strategy language. An ELAN specification consists of a set of unlabeled rewrite rules, which are applied using a fixed innermost strategy, and labeled rules, which are applied by user-defined strategies. Rewrite rules support matching modulo associativity and commutativity.

A strategy expression combines several rule labels by means of strategy operators. The application of a strategy to a term leads to a set of results. An empty set of results denotes failure. Evaluation of a term involves normalizing the term according to the unlabeled rules and then applying a strategy to it.

Strategies exist at two levels: the elementary strategies built into the language that can be used to apply labeled rules and defined strategies, which are interpreted using innermost rewriting.

An example of the use of labeled rules and elementary strategies in the definition of Knuth-Bendix completion (from [31]) is shown in Figure 13. Other applications of ELAN include constraint solving and communication protocol testing.

The identity strategy `id` succeeds and returns the singleton set containing the subject term. The failure strategy `fail` returns the empty set. The sequential composition `e1; e2` of two strategies first applies `e1`, then `e2`. There are several choice operators each with different back-tracking properties. 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)` (`iterate+(e)`) returns all possible results from iterating the strategy `e` zero (one) or more times. The operator `repeat*(e)` (`repeat+(e)`) returns the last set of results from repeatedly applying `e` until it fails. Finally, the operator `normalize(e)` normalizes a term with respect to a strategy, i.e., applying the strategy to all sub-terms until it fails for all sub-terms. Note that all other operators apply labeled rules at the root. There is no other support for term traversal using elementary strategies.

Using defined strategies language specific traversal can be defined using congruence operators.

A limited form of genericity is provided by the preprocessor mechanism of the language, which supports the definition of specification schemas. ELAN does not support language generic term traversal.

```
[Delete]    (E U {s=s} ; R)   => (E ; R)                             end
[Compose]   (E ; R U {s->t}) => (E ; R U {s->u}) if reduce(t->u) end
[Simplify] (E U {s=t} ; R)   => (E U {s=u} ; R)  if reduce(t->u) end
[Orient]    (E U {s=t) ; R)   => (E ; R U {s->t}) if s > t         end
[Collapse] (E ; R U {s->t}) => (E U {u=t} ; R)  if reduce(s->u) end
[Deduce]    (E ; R)           => (E U {s=t} ; R)  if s=t in CP(R) end

completion =>
  repeat*(repeat*(repeat*(Collapse);
                  repeat*(Compose) ;
                  repeat*(Simplify) ;
                  repeat*(Delete) ;
                  repeat*(Orient)) ;
          Deduce)
```

Fig. 13. Example ELAN rules and strategy for Knuth-Bendix completion [31].

5.10 Generating Strategies with Reflection

Maude [12] is one of the successors of algebraic specification formalism OBJ. Maude supports two kinds of rewrite rules: equations and rules. Equations are applied using an innermost strategy and rules are applied using an outermost strategy. Pattern matching in equations and rules can use matching modulo many combinations of associativity, commutivity and identity.

Maude does not support operators for the definition of strategies. Instead the language provides a meta-level in which specifications can be manipulated by reflection [13, 14].

5.11 Generic Traversal Strategies

Stratego [51, 52] is language for program transformation completely based on rewriting strategies. Stratego supports sequential programming with local backtracking, generic and specific term traversal, strategy abstraction, and first class pattern matching.

Figure 14 gives several example uses of strategies. The examples use basic operators and defined strategies discussed below. The strategies `disj-nf` and `conj-nf` define normalizations to disjunctive and conjunctive normal-form, respectively, using the rules from Figure 6. The `eval` strategy performs constant folding on propositional formulae using the standard thruth rules (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).

Sequential Programming Strategies are programs that attempt to transform terms into terms, at which they may succeed or fail. In case of success the result of such an attempt is a transformed term. In case of failure there is no result of the transformation. Strategies can be combined into new strategies by means of the following operators. The *identity* strategy `id` leaves the subject term unchanged and always succeeds. The *failure* strategy `fail` always fails. The *sequential composition* `s1 ; s2` of strategies `s1` and `s2` first attempts to

```
strategies
  disj-nf = innermost(DAOL + DAOR + DN + DMA + DMO)
  conj-nf = innermost(DOAL + DOAR + DN + DMA + DMO)

  T       = T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 +
            T11 + T12 + T13 + T14 + T15 + T16 + T17 + T18 + T19
  eval    = bottomup(repeat(T))

  desugar = topdown(try(DefI + DefE))
  impl-nf = topdown(repeat(DefN + DefA2 + DefO1 + DefE))
```

Fig. 14. Various transformations on propositonal formulae.

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

Fig. 15. Generic iteration strategies.

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. The *recursive closure* `rec x(s)` of the strategy `s` attempts to apply `s` to the entire subject term and the strategy `rec x(s)` to each occurrence of the variable `x` in `s`. 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 also fails if `s` fails. The *negation* `not(s)` succeeds (with the identity transformation) if `s` fails and fails if `s` succeeds. Two examples of strategies that can be defined with these operators are the `try` and `repeat` strategies in Figure 15

Term Traversal The combinators discussed above combine strategies that apply transformations to the root of a term. In order to apply transformations throughout a term it is necessary to traverse it. For this purpose, System S provides a *congruence* operator `C(s1,...,sn)` for each `n`-ary constructor `C`. It applies to terms of the form `C(t1,...,tn)` and applies `si` to `ti`. An example of the use of congruences is the operator `map(s)` defined in Figure 16 that applies a strategy `s` to each element of a list.

Congruences can be used to define traversals over specific data structures. Specification of generic traversals (e.g., pre- or post-order over arbitrary structures) requires more generic operators. The operator `all(s)` applies `s` to all children of a constructor application `C(t1,...,tn)`. In particular, `all(s)` is the identity on constants (constructor applications without children). The strategy `one(s)` applies `s` to one child of a constructor application `C(t1,...,tn)`; it is precisely the failure strategy on constants. The strategy `some(s)` applies `s` to some of the children of a constructor application `C(t1,...,tn)`, i.e., to at least one and as many as possible. Like `one(s)`, `some(s)` fails on constants.

Figure 16 defines various traversals based on these operators. For instance, `oncetd(s)` tries to find *one* application of `s` somewhere in the term starting at the root working its way down; `s <+ one(x)` first attempts to apply `s`, 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. Compare this to the traversal `alltd(s)`, which finds *all* outermost applications of `s` and never fails.

```
strategies
  topdown(s)      = rec x(s; all(x))
  bottomup(s)     = rec x(all(x); s)
  downup(s)       = rec x(s; all(x); s)
  downup2(s1,s2)  = rec x(s1; all(x); s2)
  oncetd(s)       = rec x(s <+ one(x))
  onebu(s)        = rec x(one(x) <+ s)
  alltd(s)        = rec x(s <+ all(x))
  sometd(s)       = rec x(s <+ some(x))
  somebu(s)       = rec x(some(x) <+ s)

  innermost(s)    = rec x(all(x); try(s; x))
```

Fig. 16. Generic traversal strategies.

Match, Build and Variable Binding The operators we have introduced thus far are useful for repeatedly applying transformation rules throughout a term. Actual transformation rules are constructed by means of pattern matching and building of pattern instantiations.

A *match* `?t` succeeds if the subject term matches with the term `t`. As a side-effect, any variables in `t` are bound to the corresponding subterms of the subject term. If a variable was already bound before the match, then the binding only succeeds if the terms are the same. This enables non-linear pattern matching, so that a match such as `?F(x, x)` succeeds only if the two arguments of `F` in the subject term are equal. This non-linear behavior can also arise across other operations. For example, the two consecutive matches `?F(x, y); ?F(y, x)` succeed exactly when the two arguments of `F` are equal. Once a variable is bound it cannot be unbound.

A *build* `!t` replaces the subject term with the instantiation of the pattern `t` using the current bindings of terms to variables in `t`. A scope `{x1,...,xn: s}` makes the variables `xi` local to the strategy `s`. This means that bindings to these variables outside the scope are undone when entering the scope and are restored after leaving it. The operation `where(s)` applies the strategy `s` to the subject term. If successful, it restores the original subject term, keeping only the newly obtained bindings to variables.

Abstraction A strategy definition `f(x1,...,xn) = s` introduces a new strategy operator `f` parameterized with strategies `x1` through `xn` and with body `s`. Such definitions cannot be recursive, i.e., they cannot refer (directly or indirectly) to the operator being defined. All recursion must be expressed explicitly by means of the recursion operator `rec`. Labeled transformation rules are abbreviations of a particular form of strategy definitions. A conditional rule `L : l -> r where s` with label `L`, left-hand side `l`, right-hand side `r`, and condition `s` denotes a strategy definition `L = {x1,...,xn: ?l; where(s); !r}`. Here, the body of the rule first matches the left-hand side `l` against the subject term, and then attempts to satisfy the condition `s`. If that succeeds, it builds the right-hand side `r`. The rule is enclosed in a scope that makes all term variables `xi` occurring freely in `l`, `s` and `r` local to the rule.

Stratego also supports pattern abstraction by means of *overlays* [49].

Generic Strategies Using the machinery of Stratego very generic strategies can be defined. The Stratego library defines a wide range of generic strategies including traversal strategies such as shown in Figure 16. 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 and work generically otherwise [50].

A problem of some generic strategies is that they lack knowledge of the computations done in their argument strategies, which may lead to too much work. For example, the innermost strategy in Figure 16 renormalizes arguments of left-hand sides of rules when they are used in the right-hand side. In [28] it is shown how this can be repaired by fusing the generic innermost strategy with its arguments.

6 Concluding Remarks

In this paper I have discussed strategies in a number of systems. The focus in the paper is on an analysis of the basic elements of languages for the specification of transformation strategies, in particular, in languages based on term rewriting. The following styles of strategy specification were discussed:

- simple tree parsing
- tree parsing with dynamic programming
- exhaustive evaluation
- traversal functions
- strategy annotations
- sequence of canonical forms
- non-deterministic sequential strategies
- reflective strategies
- generic traversal strategies

There are several issues not yet covered by this survey: implementation of transformation strategies in general purpose programming languages; constraint and goal based transformation strategies; a comparison of different paradigms of program transformation; examples of strategies for more complex program transformations.

The overview of the various systems gives rise to some questions about interesting combinations and other research topics:

Dynamic programming How can dynamic programming in the style of BURG be expressed in a more general framework of strategies while obtaining the same efficiency? Can the approach be generalized to more complex, cascading transformations? In other words, how to find the optimal sequence of transformations

Types Is there a type system that reconciles static typing with generic strategies?

Variables How to transparently deal with variable bindings and other context-sensitive issues? What is the interaction between strategies and higher-order abstract syntax?

Graphs In [30] rewrite rules on control-flow graphs are defined using temporal logic style assertions. What is the role of strategies in graph transformation?

Fusion Generic strategies parameterized with rules or other strategies often have to re-normalize/re-traverse terms. In [28] an optimization for the case of the generic definition of innermost is given. There is a general need for fusion of generic traversals.

Aspects In general, the fusion of generic strategies with rules can be seen as a form of aspect weaving [29]. Can strategies be formulated in terms of aspect-oriented programming?

Origins Origin tracking [47] for term rewriting relates the result of a normalization to the original. Applications include error messages and layout reconstruction. How to compute origins in a system with strategies? Possibly origins can be transparent to strategies. If origin information is stored in the tree, only rules need to maintain origin information.

This survey is part of a larger effort to create an overview of the theory and practice of program transformation in the online survey of program transformation [53].

Acknowledgements

I would like to thank Bernhard Gramlich and Salvador Lucas Alba for inviting me to write this paper for the Workshop on Reduction Strategies in Rewriting and Programming.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques, and tools.* Addison Wesley, Reading, Massachusetts, 1986.
2. A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.
3. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *ELAN: User Manual.* Loria, Nancy, France, v3.4 edition, January 27 2000.
4. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier.
5. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First Workshop on Rewriting Logic and Applications 1996 (WRLA'96).
6. J. M. Boyle. Abstract programming and program transformation—An approach to reusing programs. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press, 1989.
7. J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, 1997.
8. M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. Draft, March 2001.
9. M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36:209–266, 2000.
10. M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.
11. E. Chikofski and J. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 1990.
12. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89, Asilomar, Pacific Grove, CA, September 1996. Elsevier.
13. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First International Workshop on Rewriting Logic and its Applications.
14. M. G. Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language.* PhD thesis, Universidad de Navarra, Facultad de Filosofia y Letras, Pamplona, España, 1998.
15. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient and stealthy opaque constructs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 98)*, pages 184–196, San Diego CA, USA, January 1998.

16. K. Czarnecki and U. W. Eisenecker. Intentional programming. In *Generative Programming. Methods, Tools, and Applications*, chapter 11. Addison-Wesley, 2000.
17. O. de Moor and G. Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 2001. To appear, preprint available at http://web.comlab.ox.ac.uk/oucl/work/oege.demoor/papers/match.ps.gz.
18. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier, 1990.
19. A. Van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996.
20. C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *Semantics, Applications and Implementation of Program Generation (SAIG'00)*, Springer Lecture Notes in Computer Science, 2000.
21. W. J. Fokkink, J. F. T. Kamperman, and H. R. Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, January 2000.
22. M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison Wesley Longman, 1999.
23. C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.
24. C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG—fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.
25. J. A. Goguen and T. W. et. al. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International Computer Science Laboratory, March 1992.
26. T. E. H. Kienle, J. Czeranski. Exchange format bibliography. In *Workshop on Standard Exchange Format (WoSEF)*, pages 2–9, Limerick, Ireland, June 2000.
27. G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
28. P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. L. Alba, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, Utrecht, The Netherlands, May 2001. To appear.
29. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.
30. D. Lacey and O. de Moor. Imperative program transformation by rewriting. In *Compiler Construction (CC'01)*, Lecture Notes in Computer Science. Springer-Verlag, April 2001.
31. P.-E. Moreau. *Compilation de règles de réécriture et de stratégies non-déterministes*. PhD thesis, L'Université Henri Poincaré-Nancy 1, June 22 1999.
32. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
33. G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski, editor, *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, volume 1, pages 810–827, Cambridge, Mass., USA, 1988. MIT Press.
34. J. M. Neighbors. *Software Construction Using Components*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, 1980.

35. J. M. Neighbors. *Draco 1.2 Users Manual.* Department of Information and Computer Science, University of California, Irvine, Irvine, CA, USA, June 1983. Available from http://www.bayfronttechnologies.com/manual.htm.
36. J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–573, September 1984.
37. K. Ogata and K. Futatsugi. Implementation of term rewritings with the evaluation strategy. In *Proceedings 9th Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*, pages 225–239, Southampton, 1997. Springer-Verlag.
38. T. J. Parr. SORCERER reference. In *Language Translation Using PCCTS and C++. A Reference Guide*, chapter 4, pages 161–199. Automata Publishing Company, 1993. Available at http://www.antlr.org/buybook.html.
39. T. J. Parr. An overview of SORCERER: A simple tree-parser generator. http://www.antlr.org/papers/sorcerer.ps, April 1994.
40. S. Peyton Jones, J. Hughes, et al. Report of the programming language Haskell98. a non-strict, purely functional language, February 1999.
41. F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 199–208. ACM, 1988.
42. A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proceedings of the 5th International Conference on Mathematics of Programme Construction (MPC2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255, Ponte de Lima, Portugal, July 2000. Springer-Verlag.
43. T. A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.
44. D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.
45. M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
46. J. van der Pol. Just-in-time: On strategy annotations. In *Proceedings of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, 2001. (To appear).
47. A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5–6):523–546, 1993.
48. E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.
49. E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.
50. E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.
51. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, Lecture Notes in Computer Science. Springer-Verlag, May 2001. To appear.

52. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).
53. E. Visser et al. The online survey of program transformation. www.program-transformation.org, 2000–2001.
54. P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 307–313, Munich, January 1987. ACM.