

Program Transformation: What, How, and Why

Victor L. Winter *
University of Nebraska at Omaha
Department of Computer Science
vwinter@mail.unomaha.edu

Contents

1	What: The Manipulation of Complex Values	2
2	How: Equational Reasoning – the Essence of Program Transformation	3
2.1	Equational Reasoning: A technique for mathematical manipulation	4
2.2	The Mechanism of Equational Reasoning	5
2.2.1	Variables and Matching	5
2.2.2	Equation Orientation, Confluence and Termination	6
2.2.3	Rule Extensions and Application	7
2.2.4	Program Fragments as “expressions”	7
2.3	Example: A Pseudo-compiler for Imp	8
2.4	Program Transformation Frameworks	10
2.4.1	Strategic Combinators	11
2.4.2	Traversals	12
2.4.3	Strategic Frameworks	12
3	Why: Applications	12
3.1	Transformations that Shift Between Semantic Models	13
3.2	Transformations that Remain within a Single Semantic Model	14
3.2.1	Refactoring	14
3.2.2	The Evaluation of λ -expressions	15
4	Summary and Conclusion	16

Abstract

Transformation can be viewed as a philosophy on how to achieve change. A rigorous treatment of transformation has its roots in *equational reasoning* – the idea that equals can be substituted for equals. This article explores *transformation* as it applies to the manipulation of software.

*This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy. Victor Winter was also partially supported by NSF grant number CCR-0209187.

1 What: The Manipulation of Complex Values

A typical computer program consists of a sequence of instructions that manipulate values belonging to a variety of *simple* data types. In this context, a data type is considered to be simple if its values have a simple syntactic structure. Integers, reals, Booleans, strings, and characters are all examples of simple data types. In contrast, when viewed as a value, the sequence of characters that make up a program written in a high-level language such as Java or C++ can be seen as having a highly complex syntactic structure.

Informally speaking, a good litmus test for determining whether a particular value is simple is to consider the complexity of user-defined methods capable of reading in such a value from a file, storing this value internally within a program, and writing this value to a file. Thinking along these lines reveals that typical computer languages provide I/O support for simple types (e.g., `getc`, `read`, `input1`, `inputN`, `put`, `print`, `write`, etc.) as well as primitive support for basic operations on these types (e.g., equality comparison, relational comparisons, addition, subtraction, and so on).

A similar level of support is generally not provided for values having syntactic structures that cannot be directly modeled in terms of simple values. Thus, as the structure of the data becomes more complex, a greater burden is placed on the programmer to develop methods capable of performing desired operations (e.g., I/O, equality comparison, internal representation, and general manipulations). In the limit, the techniques employed for structure recognition include the development of domain-specific parsers, reuse of general purpose context-free parsers such as LL, LALR, LR parsers [2], and even state-of-the-art parsers such as Scannerless Generalized LR (SGLR) parsers [26][7]. The values constructed by these tools are typically output using sophisticated algorithms such as abstract pretty printers [9][24].

Parsers such as LL, LALR, LR, and SGLR parsers all ultimately make use of powerful parsing algorithms for recognizing the structure of a sequence of symbols. From a theoretical perspective, these parsing algorithms are capable of recognizing the class of languages known as context-free languages. This class of languages is interesting because it represents the most complex class that can be efficiently recognized by a computer using general purpose algorithms. The syntactic structure of modern programming languages typically fall in the class of context-free languages or slight variations thereof [15].

Figure 1 gives an example of an extended-BNF grammar fragment describing the syntactic structure of a simple imperative language we will call *Imp*. The directives *%LEFT_ASSOC ID* and *%PREC ID* are used to declare and assign precedence and associativity to operations and productions in the grammar (for more on precedence and associativity see [2]). These assignments allow portions of the grammar that would otherwise be ambiguous to be uniquely parsed. Informally summarized, the language described by the grammar fragment defines an *Imp* program as consisting of a single block containing a statement list. In turn, a statement list consists of zero or more labeled statements. A label may be optionally associated with a statement. A statement can either be a block, one of three different kinds of if-statements, a while-loop, an assignment, a goto-statement, or a statement called *skip* whose execution doesn't do anything (i.e., *skip* is a no-op). Programs written in this language can be parsed using an LALR parser that has been extended with associativity and precedence.

As a result of their context-free roots, the structure of character sequences corresponding to typical computer programs can be modeled in terms of a tree structure (also known as a term structure). Tree structures come in two basic flavors: parse trees which literally reflect the structure described by the context-free grammar used to define the programming language, or abstract syntax trees which capture the essence of the structure described by the context-free grammar (for more on extended-BNF grammars and abstract syntax see [25]). More compact internal representations such as directed acyclic graphs (DAGs) are also possible, but a discussion of these lies beyond the

	%LEFT_ASSOC L1.	%LEFT_ASSOC L2.	%LEFT_ASSOC L3.	%LEFT_ASSOC L4.	%LEFT_ASSOC L5.
pgm	::=	{ stmt_list }			
stmt_list	::=	labeled_stmt ; stmt_list ϵ			
labeled_stmt	::=	label stmt			
label	::=	id : ϵ			
stmt	::=	block cond loop assign jump continue			
block	::=	{ stmt_list }			
cond	::=	if E labeled_stmt			
cond	::=	if E then block		%PREC L1	
cond	::=	if E then block else block		%PREC L2	
loop	::=	while E do block			
assign	::=	id = E			
jump	::=	goto label			
continue	::=	skip			
E	::=	E op E (E) !(E) base			
op	::=	< > <= >= = !=		%PREC L3	
op	::=	+ -		%PREC L4	
op	::=	* div mod		%PREC L5	
base	::=	id integer			
id	::=	identifier			

Figure 1: A grammar fragment of a simple imperative language called Imp

scope of this article.

2 How: Equational Reasoning – the Essence of Program Transformation

Program transformation concerns itself with the manipulation of programs. Conceptually speaking, a (program) transformation system accepts a *source program* as its input data and produces a transformed program known as a *target program* as its output data. Thus, a transformation system treats programs in much the same way that traditional programs treat simple data. In general, systems that share this view of programs-as-data are called *meta-programming systems*. A compiler is a classic example of a meta-programming system.

In spirit, the goal in program transformation is to manipulate programs using techniques similar to the techniques used by mathematicians when they manipulate expressions. For example, in mathematics, the expression $x \wedge true$ can be simplified to x . Similarly in Java, the sequence of assignments $x = 5; x = x$ can be simplified to the single assignment $x = 5$. In Boolean algebra, the expression $e1 \vee e2$ is equivalent to $e2 \vee e1$ for any arbitrary Boolean expressions $e1$ and $e2$. However, in Java, Boolean expressions are conditionally evaluated¹ and as a result $e1 || e2$ is not equivalent to $e2 || e1$ (consider the evaluation of the Boolean expression $true || 4/0 < 5$). On the other hand, in Java a conditional statement of the form *if (BE) stmt₁; else stmt₂;* is equivalent

¹This form of evaluation is also referred to as short-circuiting.

to *if (!BE) stmt₂; else stmt₁*; for any Java Boolean expression *BE* and Java statements *stmt₁* and *stmt₂*. Having seen a few examples of manipulation let us take a more detailed look at how mathematical expressions can be manipulated in general through a process known as *equational reasoning*.

2.1 Equational Reasoning: A technique for mathematical manipulation

In mathematics there are axioms (i.e., laws) and theorems stating how expressions of a certain type (e.g., Boolean expressions) can be manipulated. Axioms and theorems are oftentimes given in the form of equations relating two syntactically distinct expressions. Figure 2 gives a standard set of axioms defining a Boolean algebra.

<p>Commutativity</p> $or(x, y) = or(y, x)$ $and(x, y) = and(y, x)$	<p>Identity</p> $or(x, false) = x$ $and(x, true) = x$
<p>Distributivity</p> $or(x, and(y, z)) = and(or(x, y), or(x, z))$ $and(x, or(y, z)) = or(and(x, y), and(x, z))$	<p>Complement</p> $or(x, not(x)) = true$ $and(x, not(x)) = false$

Figure 2: The standard axioms for a Boolean algebra

The axioms for Boolean algebra provide us with the basis for manipulating Boolean expressions. In mathematics, when manipulating a mathematical expression a common goal is the *simplification* of that expression. In math classes, problems are often given where the goal is to simplify an expression until it can be simplified no further. This activity is referred to as *solving* the expression and the simplified form of the expression is called the *answer*. In the context of equational reasoning such an *answer* is called a *normal form*. For example, the *normal form* of $7 * 7 + 1$ is 50. In this article, we will use the terms *rewriting* and *simplification* interchangeably.

In addition to expression simplification, in mathematics one is also interested in knowing whether one expression is equal to another expression. This activity is known as *theorem proving*. Theorems have the general form: $e1 = e2$ *if cond*, where *cond* defines the conditions under which $e1 = e2$ holds. In the degenerative case where $e1 = e2$ always holds one may drop the conditional portion and simply write the theorem as $e1 = e2$.

Suppose that one is interested in knowing whether $or(b, b) = b$ is a theorem, where $or(b, b)$ is the prefix form of the Boolean expression $b \vee b$. How does one go about proving such a theorem? One approach for proving a theorem of the form $e1 = e2$ is to separately try to *rewrite* $e1$ and $e2$ into their normal forms and then compare the results. A variation of this idea is to pick whichever term $e1$ or $e2$ is more complex and rewrite it in the hopes that it can be simplified to the other term. Having said that, we will view the proof of $or(b, b) = b$ in terms of a simplification problem. In particular, we are interested in rewriting the expression $or(b, b)$ to b , which conveniently already happens to be in its normal form, thereby proving the theorem $or(b, b) = b$. The proof of $or(b, b) = b$ is shown in Figure 3. An important thing to note about the sequence of “simplifications” that are applied to $or(b, b)$ is that they are anything but simple. It turns out that in the context of first-order logic, there is no universal definition for the notion of *simplification* that can be used to prove all theorems. Indeed, it is well known that theorem proving in the realm of first-order logic is in fact undecidable. The implications of this observation is that the complete automation of Boolean simplification is not realistic.

Operationally, the simplifications shown in Figure 3 are accomplished through a process known as *equational reasoning* which is based on equational logic [3]. Informally stated, equational reasoning is the notion that “equals may be substituted for equals”. The axioms of Boolean algebra shown in Figure 2 provide us with an initial set of equal quantities in the form of equations, and it is instances of these axioms that are used in the proof shown in Figure 3.

$or(b, b)$	Given
$and(or(b, b), true)$	<i>Identity</i> : $x \rightarrow and(x, true)$ where x is $or(b, b)$
$and(or(b, b), or(b, not(b)))$	<i>Complement</i> : $true \rightarrow or(x, not(x))$ where x is b
$or(b, and(b, not(b)))$	<i>Distributivity</i> : $and(or(x, y), or(x, z)) \rightarrow or(x, and(y, z))$ where x is b , y is b and z is $not(b)$
$or(b, false)$	<i>Compliment</i> : $and(x, not(x)) \rightarrow false$ where x is b
b	<i>Identity</i> : $or(x, false) \rightarrow x$ where x is b

Figure 3: An example of axiom-based manipulations of Boolean expressions

Equational reasoning is a cornerstone of mathematics and is an indispensable tool at the mathematician’s disposal when it comes to reasoning about expressions. In theory, the concepts and mechanisms underlying equational reasoning should also be adaptable to reason about and manipulate programs. Just as in mathematics, in computer science there are axioms and theorems stating how program structures belonging to a given language relate to one another. Realizing this, our original definition of *program transformation* can be refined as follows:

Program transformation involves the discovery and development of suitable axioms and theorems and their application to programs in accordance with the laws of equational logic to achieve a particular goal.

2.2 The Mechanism of Equational Reasoning

In order to consider manipulating programs in the way mathematicians manipulate expressions it is helpful to first analyze and abstract the techniques and concepts underlying equational reasoning. In addition, we are interested in knowing the extent to which various techniques and processes can be automated. Ideally, we are aiming for a fully automated system that when presented with a program and a goal (e.g., simplification) will produce an output program satisfying that goal.

2.2.1 Variables and Matching

In equational reasoning, the *variable* plays an important role. For example, the axioms in Figure 2 make use of the variables x , y , and z . Variables allow equations to be written that capture general relationships between expression structures. *Matching* [3] is an activity involving variables that is very important in equational reasoning. Let e denote an expression we are interested in manipulating, and let $e_1 = e_2$ denote the equation we are considering using in order to manipulate e . Matching allows us to determine whether e is an instance of e_1 or e_2 . For example, in the proof in Figure 3 it is possible to rewrite $or(b, b)$ to $and(or(b, b), true)$ using the equation $and(x, true) = x$ and realizing that $or(b, b)$ is an instance of x (i.e., the variable x can denote a quantity like $or(b, b)$). Similarly, it is possible to rewrite the expression $or(b, and(b, not(b)))$ to $or(b, false)$ by using the equation $and(x, not(x)) = false$ and realizing that the sub-expression $and(b, not(b))$ is an instance of $and(x, not(x))$.

Let e denote an expression that may contain one or more variables and let t denote an expression containing no variables. We will write $e \ll t$ to denote the attempt to *match* e with t . We will refer to $e \ll t$ as a *match equation*. A match equation is a Boolean valued test that either succeeds or fails. If a match equation succeeds then it means that t is an instance of e . More specifically, this means that there exist values that when substituted for the variables in e will produce the expression t . For example, if we substitute b for x in the expression $and(x, not(x))$ we get $and(b, not(b))$, thus $and(x, not(x)) \ll and(b, not(b))$ succeeds under the substitution $x \mapsto b$. Substitutions are abstractly denoted by the symbol σ . The act of replacing the variables in an expression e as defined by σ is known as *applying* the substitution σ to e and is written $\sigma(e)$.

Matching related concepts have been heavily researched. Under suitable conditions it is appropriate to use more powerful algorithms to construct an expression that is an instance of two other expressions. These algorithms include *unification* [21], *AC-matching* [12], *AC-unification* [17], and even *higher-order unification and matching* [11].

2.2.2 Equation Orientation, Confluence and Termination

Given an expression t a crucial aspect of equational reasoning is how one makes the decision regarding which equation should be used to simplify t or one of its sub-expressions. In the realm of rewriting, the complexity of the decision making process has been simplified by orienting equations. For example, instead of writing $e_1 = e_2$, one would write $e_1 \rightarrow e_2$. An oriented equation of the form $e_1 \rightarrow e_2$ is called a *rewrite rule*. The orientation $e_1 \rightarrow e_2$ constrains the equational reasoning process to the replacement of instances of e_1 by instances of e_2 and not the other way around².

Orienting equations into rewrite rules greatly simplifies the task of deciding which rewrite rule should be applied to a given term. However, equation orientation does not eliminate the decision altogether. In general, expressions still exist to which two or more competing rules apply (see Section 2.2.3 for more details on rule application). Under such conditions we say that the rules *interfere* with one another. The simplest example of a pair of interfering rules are two rewrite rules having identical left-hand sides (e.g., $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$). Ideally, we would like to have a set of rules that do not interfere with each other, or at least know that if rules do interfere with one another the interference somehow doesn't matter. A consequence of the notion of "interference not mattering" is that the normal form for an expression, when it exists, must be unique. In general, rule sets having the property of "interference not mattering" are said to be *confluent* or equivalently *Church-Rosser* [1][3]. Formally, the *Church-Rosser* property is defined as: $e_1 \xrightarrow{*} e_2 \Rightarrow e_1 \downarrow e_2$. Informally, this means that expressions that are equal can always be *joined* through the application of rewrite rules (i.e., oriented equations) in the (Church-Rosser) rule set. In other words, given a rule set R , we say that two expressions can be *joined* if they both can be rewritten to the same expression using only the rewrite rules found in R .

An important result concerning the confluence/Church-Rosser property is that it is possible to mechanically check whether a rule set possesses this property. It is also possible in certain cases to convert a rule set that is not confluent into an equivalent rule set that is confluent [3].

Confluence is a highly desirable property for a rule sets to possess because it implies that the decision of which order rules should be applied during the course of an equational reasoning session is immaterial. Thus, the algorithm driving the equational reasoning process is trivial, one simply applies rules where ever and whenever possible secure in the knowledge that the rewriting process will always arrive at the same *normal form*, when it exists.

²A discussion of the techniques used to decide how equations should be oriented lies beyond the scope of this paper.

When does a normal form not exist? Given a confluent rule set, the only circumstances under which a normal form does not exist is if the rule set is nonterminating. For example, consider the rule set consisting of the single rule $x \rightarrow f(x)$. This rule set is trivially confluent but is nonterminating and therefore produces no normal forms. Using this rule set to “simplify” the expression b will yield the nonterminating sequence of rewrites $b \rightarrow f(b) \rightarrow f(f(b)) \rightarrow \dots$. A rule set is said to be *terminating* if every simplification sequence eventually produces a normal form. The combination of confluence and termination let us conclude that all expressions have a normal form and that their normal forms are unique.

In general, the problem of showing that a rule set is terminating is undecidable. However, in practice one can often show that a particular rule set is terminating. Because of the highly desirable properties of rule sets that are confluent and terminating, the termination problem is a heavily researched area in the field of rewriting [3].

2.2.3 Rule Extensions and Application

The basic notion of a rewrite rule can be extended in two important ways. In the first extension allows a label to be associated with a basic rewrite rule. The result is called a *labeled rewrite rule*. Labeled rewrite rules typically have the form $label : lhs \rightarrow rhs$ where lhs and rhs are expressions. A transformation system supporting labeled rewrite rules, allows the option of labeling rewrite rules and treats a reference to a label as a short-hand for a reference to the rule.

In the second extension, a labeled rewrite rule can be extended with a condition. The result is called a *labeled conditional rewrite rule*. Conditions can take on a number of forms but all ultimately can be understood as a Boolean condition that enables or prohibits a rewrite rule from being applied. Consider the rule $x/x \rightarrow 1$ if $x \neq 0$. In this article, a labeled conditional rewrite rule has the form $label : lhs \rightarrow rhs$ if *condition*. We will also only consider a restricted form of condition consisting of Boolean expressions involving *match equations* as defined in Section 2.2.1.

Let r denote an arbitrary rewrite rule and let e denote an expression. If r is used as the basis for performing a manipulation of e we say that r is applied to e , and this is what we mean when we say *rule application*. More specifically, when using a conditional rewrite rule of the form $lhs \rightarrow rhs$ if *cond* to simplify an expression t one first evaluates the Boolean expression $lhs \ll t \wedge cond$. If this Boolean expression evaluates to *true* and produces the substitution σ , then t is *rewritten* to rhs' , where $rhs' = \sigma(rhs)$ is the instance of rhs obtained by applying the substitution σ to the expression rhs .

2.2.4 Program Fragments as “expressions”

Thus far, we have given an overview of the mechanisms underpinning rewriting. However, we have not said much about notations for describing expressions. When manipulating Boolean expressions, the choice of notation is fairly straightforward. One can, for example, write a Boolean expression in infix form $e_1 \vee e_2$ or in prefix form $or(e_1, e_2)$. How do these ideas translate to program structures? One possibility is to express code fragments in prefix form. However, there are some disadvantages to such an approach. One disadvantage is that there is some notational complexity associated with prefix forms since this is not how we write programs in general. This conceptual gap holds in the realm of Boolean algebra as well. For example, most readers will probably find $x \vee y \wedge z$ to be more readable than $or(x, and(y, z))$. This problem is amplified as the complexity of the structure expressed increases (and code fragments can have a complex structure). To address the comprehensibility problem we will express code fragments in an infix form that we call a *parse expression* [30][32]. A parse expression is essentially a short-hand for a parse tree and assumes

that the syntax of the programming language has been defined by an extended-BNF. In general a parse expression has the form $B[\alpha']$ where B is a nonterminal in the grammar and the derivation $B \stackrel{\dagger}{\Rightarrow} \alpha$ is possible. The difference between α as it occurs in the derivation and α' as it occurs in the parse expression is that in α' all nonterminal symbols have been subscripted, making them *variables*. In particular, when we say *variable* we mean a symbol that can participate in matching as described in Section 2.2.1.

Let us consider the grammar fragment for Imp shown in Figure 1. The parse expression $assign[id_1 = E_1]$ denotes a parse tree whose root is the nonterminal $assign$ and whose leaves are id_1 , $=$, and E_1 . Since id_1 and E_1 are variables this parse expression denotes the most general form of an assignment statement. The expression $assign[id_1 = E_1 op_1 E_2]$ denotes a less general form of an assignment in which an identifier id_1 is bound to an expression $E_1 op_1 E_2$. That is, an expression containing a least one binary operator.

Matching works for parse expressions just as would be expected. For example, the match equation $assign[id_1 = E_1] \ll assign[x = 5 + 4]$ succeeds with the substitution $id_1 \mapsto id[x]$ and $E_1 \mapsto E[5 + 4]$. Similarly, the match equation $assign[id_1 = E_1 op_1 E_2] \ll assign[x = 5 + 4]$ also succeeds with the substitution $id_1 \mapsto id[x]$, $E_1 \mapsto E[5]$, and $E_2 \mapsto E[4]$. We are now ready to look at a more concrete example of program transformation.

2.3 Example: A Pseudo-compiler for Imp

A compiler takes a source program as input and produces an assembly program as output. As such, a compiler is a meta-programming system. In this section, we look at an example of how an Imp program can be partially compiled via rewriting. The goal in our example is to take an Imp *source program* and transform it into an Imp *target program*. We claim, without proof, that the rewrite rules presented for accomplishing this are both confluent and terminating. The normal form of an Imp *source program* is an Imp *target program*, and can be obtained by the exhaustive application of the labeled conditional rewrite rules shown in Figure 6.

In order to be considered a *target program* an Imp program should satisfy the following properties:

- All expressions in the target program should be *simple expressions*. An expression is a *simple expression* if it satisfies one of the following properties: (1) the expression consists solely of a base value (i.e., either an integer or an identifier), (2) the expression consists of a binary operation involving two base values (e.g., $15 + 27$), or (3) the expression consists of a unary operation on a base value (e.g., $!(x)$). All other expressions are not simple.
- A target program may contain no *while-loops*.
- A target program may contain no *if-then* or *if-then-else* statements. Note that this makes the *if-statement* the only remaining conditional construct.
- The Boolean expression associated with the *if-statement* must be an identifier (e.g., it may not be an expression of the form $e1 op e2$).

Due to their simple structure, Imp target programs are similar to assembly programs. In fact, Imp target programs are just one step away from assembly programs and can be transformed into assembly programs on a statement by statement basis. Figure 4 gives an example of how an assignment statement can be directly transformed into a sequence of assembly instructions.

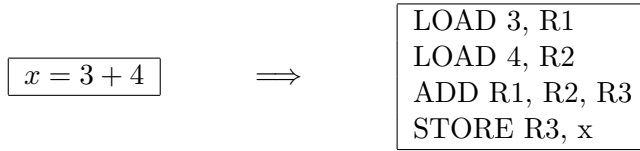


Figure 4: An example of how an assignment statement in an Imp target program can be transformed into a sequence of assembly instructions

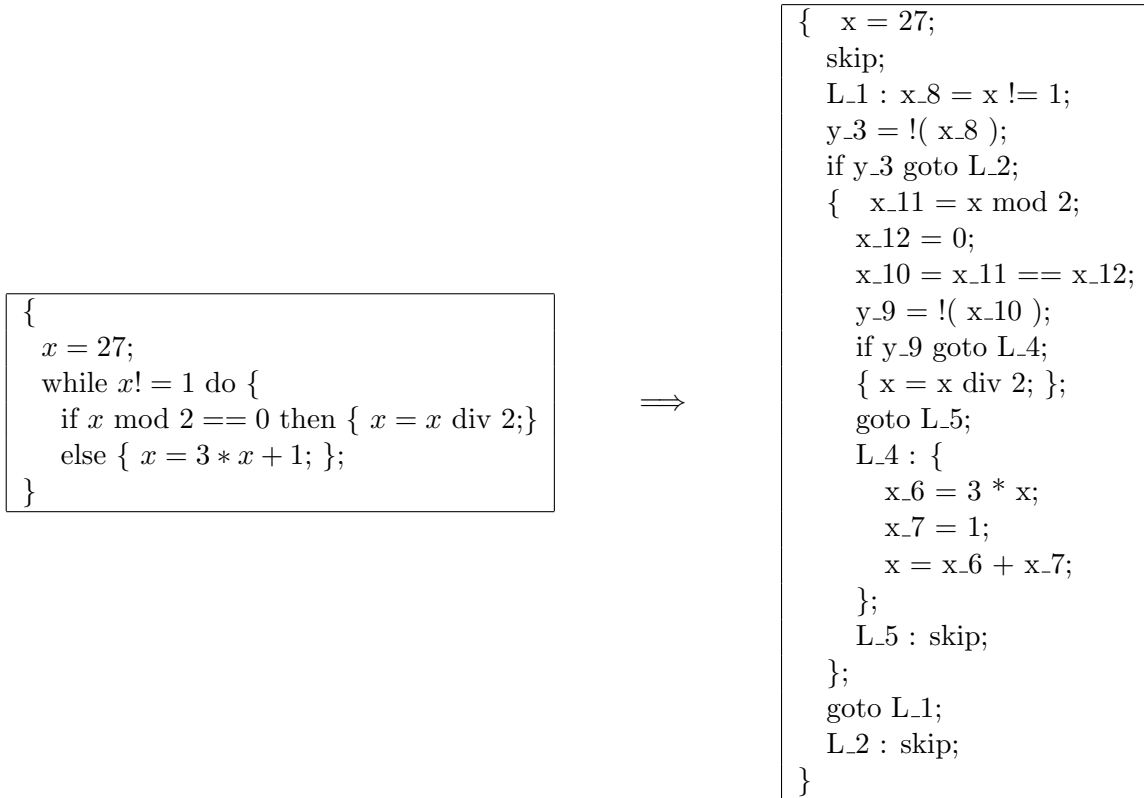


Figure 5: An Imp source program and an equivalent Imp target program

We hope the reader is convinced by this concrete example that the bulk of the general transformation from Imp target programs to assembly code is straightforward. Thus, we return our attention to the problem of transforming Imp source programs into Imp target programs.

Figure 5 shows an Imp source program and the target program that is obtained after applying the labeled conditional rewrite rules given in Figure 6. In Figure 6, the rewrite rules *assign_simplify1*, *assign_simplify2*, and *assign_simplify3* collectively account for the three cases that need to be considered when simplifying an expression in the context of an assignment statement. The rule *assign_simplify1* is a conditional rule that removes (unnecessary) outermost parenthesis from an expression. The rule *assign_simplify2* transforms the assignment of an identifier to a negated expression into a sequence of two assignment statements, provided the negated expression is not a base value. For example, the assignment $x = !(3 < 4)$ will be transformed to $x_1 = 3 < 4$; $x = !(x_1)$ where x_1 is a *new* identifier. Notice that in order to carry out this kind of manipulation, one must have the ability to generate a *new* (heretofore unused) identifier. In the rewrite rules shown, this functionality is realized by the function *new* which we do not discuss

further in this article³. And lastly, note that without the conditional check $\neg(E_1 \ll E[\llbracket base_1 \rrbracket])$ the rule *assign_simplify2* would be nonterminating.

The rule *assign_simplify3* transforms an assignment statement containing a non-simple expression (e.g., an expression containing two or more binary operators) into a sequence of three assignment statements. For example, the assignment $x = 4 + 5 * 6 * 7$ would be rewritten into the assignment sequence $x_1 = 4; x_2 = 5 * 6 * 7; x = x_1 + x_2$. Notice that the assignment $x_2 = 5 * 6 * 7$ still contains a complex expression and will again be simplified by the *assign_simplify3* rule. In the rule *assign_simplify3* the parse expression $stmt_list[id_1 = E_2 op_1 E_3; stmt_list_1]$ denotes a statement list whose first statement is the assignment of the form $id_1 = E_2 op_1 E_3$. Analysis of the problem shows that matching this structure is a necessary but not sufficient condition to ensure that an expression is not simple. In order for an expression to be not simple it must also not be the case that both E_2 and E_3 are *base* structures. Formally, this property is captured in the conditional portion of *assign_simplify3* by the Boolean expression $\neg(E_2 \ll E[\llbracket base_2 \rrbracket] \wedge E_3 \ll E[\llbracket base_3 \rrbracket])$. The remaining portion of the condition $id_2 \ll new \wedge id_3 \ll new$ is responsible for binding the variables id_2 and id_3 to *new* identifier names (e.g., $id_2 \mapsto id[\llbracket x_1 \rrbracket]$).

The remaining rules in Figure 6 make use of notational constructs similar to those we have just discussed. The rules *jump1*, *jump2*, and *jump3* are respectively responsible for rewriting *if-then* statements, *if-then-else* statements, and *while* loops into equivalent sequences consisting of *if-statements*, *labels*, *goto* statements, and *skip* statements. Here the *skip* statement is used to provide a point, beyond a given block, to which a *goto* can jump. In many cases, additional optimizing transformations can be applied to remove unneeded skip statements. However, the *skip* statement cannot be removed entirely (consider the case where the last portion of a program is a block that one wants to jump over).

And lastly, the *simplify-if* rule makes sure that the Boolean condition associated with an *if-statement* consists of a base value.

2.4 Program Transformation Frameworks

Section 2.2.2 mentioned that confluence and termination are highly desirable properties for rule sets because the problem of deciding which rule to apply then becomes immaterial. Unfortunately, when transforming programs it is often the case that rewrite rules are created that are neither confluent nor terminating and cannot be made so. Under these conditions, if transformation is to succeed, explicit control must be exercised over when, where, and how often rules are applied within a term structure. A specification of such control is referred to as a *strategy*, and systems that provide users with constructs for specifying control are known as *strategic programming systems*.

The control mechanisms in a strategic programming system fall into two broad categories: *combinators* and *traversals*. The computational unit in a rewrite system is the *rewrite rule*. Similarly, the computational unit in a strategic programming system is the *strategy*. A strategy can be inductively defined as follows:

- A rewrite rule is a strategy.
- A well-formed expression consisting of strategies, combinators, and traversals is a strategy.

Of central importance to a framework exercising explicit control over the application of rules is the ability to observe the outcome of the application of a rule to a term. Specifically, in order to exercise control a system needs to be able to answer the question: “Did the application of rule r to

³The ability to generate a *new* identifier name is supported by most program transformation systems.

<i>assign_simplify1</i>	:	$assign\llbracket id_1 = (E_1) \rrbracket \rightarrow assign\llbracket id_1 = E_1 \rrbracket$
<i>assign_simplify2</i>	:	$stmt_list\llbracket label_1 id_1 = !(E_1) ; stmt_list_1 \rrbracket \rightarrow$ $stmt_list\llbracket label_1 id_2 = E_1 ; id_1 = !(id_2) ; stmt_list_1 \rrbracket$ if $\neg(E_1 \ll E\llbracket base_2 \rrbracket) \wedge id_2 \ll new$
<i>assign_simplify3</i>	:	$stmt_list\llbracket label_1 id_1 = E_2 op_1 E_3 ; stmt_list_1 \rrbracket \rightarrow$ $stmt_list\llbracket label_1 id_2 = E_2 ; id_3 = E_3 ; id_1 = id_2 op_1 id_3 ; stmt_list_1 \rrbracket$ if $\neg(E_2 \ll E\llbracket base_2 \rrbracket) \wedge E_3 \ll E\llbracket base_3 \rrbracket) \wedge id_2 \ll new \wedge id_3 \ll new$
<i>jump1</i>	:	$stmt_list\llbracket label_1 if E_1 then block_1 ; stmt_list_1 \rrbracket \rightarrow$ $stmt_list\llbracket label_1 if !(E_1) goto id_1 ; block_1 ; id_1 : skip ; stmt_list_1 \rrbracket$ if $id_1 \ll new$
<i>jump2</i>	:	$stmt_list\llbracket label_1 if E_1 then block_1 else block_2 ; stmt_list_1 \rrbracket \rightarrow$ $stmt_list\llbracket label_1 if !(E_1) goto id_1 ;$ $block_1 ; goto id_2 ;$ $id_1 : block_2 ;$ $id_2 : skip ; stmt_list_1 \rrbracket$ if $id_1 \ll new \wedge id_2 \ll new$
<i>jump3</i>	:	$stmt_list\llbracket label_1 while E_1 do block_1 ; stmt_list_1 \rrbracket \rightarrow$ $stmt_list\llbracket label_1 skip ;$ $id_1 : if !(E_1) goto id_2 ;$ $block_1 ; goto id_1 ;$ $id_2 : skip ; stmt_list_1 \rrbracket$ if $id_1 \ll new \wedge id_2 \ll new$
<i>simplify_if</i>	:	$stmt_list\llbracket label_1 if E_1 labeled_stmt_1 ; stmt_list_1 \rrbracket \rightarrow$ $stmt_list\llbracket id_1 = E_1 ; label_1 if id_1 labeled_stmt_1 ; stmt_list_1 \rrbracket$ if $\neg(E_1 \ll E\llbracket base_1 \rrbracket) \wedge id_1 \ll new$

Figure 6: Rewrite rules capable of transforming Imp source programs into equivalent target programs

term *t* *succeed* or *fail*?” In summary then, a strategic programming system can be thought of as a rewriting system that has been extended with mechanisms for explicitly controlling the application of rules where the notion of *failure* plays a central role.

2.4.1 Strategic Combinators

A *combinator* is an operator (generally unary or binary) that can be used to compose one or more *strategies* into a new *strategy*. Let s_1 and s_2 denote two strategies. Typical combinators include

- *sequential composition* denoted $s_1; s_2$. The application of $s_1; s_2$ to a term t will first apply s_1 to t and then apply s_2 to the result.
- *left-biased choice* denoted $s_1 <+ s_2$. When applied to a term t , the strategy $s_1 <+ s_2$ will first try to apply s_1 to t , if that succeeds and produces the result t' , then t' is the result of applying $s_1 <+ s_2$ to t . Otherwise, s_2 is applied to t . If this application succeeds and produces t'' as its result, then t'' is the result of applying $s_1 <+ s_2$. However, if the application of s_2 to t fails then the application of $s_1 <+ s_2$ is said to fail.

- *right-biased choice* denoted $s_1 +> s_2$. The strategy $s_1 +> s_2$ is equivalent to $s_2 <+ s_1$.
- *nondeterministic choice* denoted $s_1 + s_2$. If both s_1 and s_2 can be applied to a term t then s_1 or s_2 is nondeterministically chosen and applied to t . If only one strategy can be applied that strategy is selected, and if both strategies do not apply then the application of $s_1 + s_2$ to the term t fails.

2.4.2 Traversals

The combinators described in Section 2.4.1 provide the ability to discriminate and sequence the application of strategies to a term. When a strategy contains a combinator, the application of that strategy to a term is defined with respect to the structure of the strategy, irrespective of the structure of the term. In contrast, a *traversal* focuses on the structure of the term, but does not consider the structure of the strategy. Broadly speaking, a traversal specifies the order in which a term and its sub-terms are visited. Thus, a traversal can be understood as a mechanism for sequencing term structures. Typically, when a term is visited some action is performed like the application of a strategy to the term.

Some traversals capture sequencing notions that are broadly applicable across a wide range of applications. Such traversals are called *generic traversals*. A typical and very useful generic traversal is one that performs a top-down left-to-right traversal of a tree structure and uniformly applies a given strategy to all sub-trees encountered. Another generic traversal is one that performs a bottom-up left-to-right traversal of a tree structure. And a third generic traversal is one where the traversal is stopped after the first successful application of a given strategy. Other generic traversals have been identified in the literature [10][19][30].

In some cases, the notion of generic traversal has direct analogies with traditional models of computation. For example, a top-down (outside-in) approach to evaluation corresponds to lazy evaluation style where functions are applied to arguments without (first) evaluating the arguments. In contrast, a bottom-up (inside-out) approach corresponds to a strict evaluation where the arguments to functions are evaluated before functions are applied.

2.4.3 Strategic Frameworks

In addition to the combinators and traversals discussed, strategic programming frameworks may contain a variety of additional features. These features can include (1) the ability to create rewrite rules and strategies dynamically [28][30][32], (2) the ability to define strategy application via *congruences* [6][10], and (3) constructs that allow user-defined generic traversals to be created [28][30].

ELAN [6], TL [30], the ρ -calculus [10], the S'_γ -calculus, and Stratego [29] are examples of strategic programming frameworks. Of these frameworks, ELAN and Stratego have implementations, and a dialect of TL is implemented in a system called HATS [14].

3 Why: Applications

Abstractly, program transformation system can be viewed as a system that transforms a *source program* into a *target program*. In [27] an excellent overview is given of a wide variety of software related activities that can be approached from a transformation-oriented perspective. Activities are broadly classified as either belonging to the category of *rephrasing* or *translation*. In this section

we present a taxonomy similar to the one given in [27] but with a greater emphasis placed on semantics. In particular, our taxonomy is motivated by the relationship between the semantic models necessary to understand the source and target programs. Within this classification system we identify seven major bi-directional goals of program transformation:

- *clarity* – This goal focuses on separation and encapsulation of functional and behavioral concerns.
- *efficiency* – This goal focuses on changing the resource usage of an executing program. Resources of primary concern are time and space.
- *computability* – This goal focuses on the translation between non-computable and computable program representations. Technically speaking, the goal of a compiler is to take a source program that cannot be directly executed on a computer and translate it into a target program that can be executed on a computer. In most cases, this goal involves moving between semantic models at two different levels of abstraction.
- *simplicity* – This goal focuses on transforming a source program to a target program where the semantic model for the source program is either a subset or superset of the semantic model of the target program.
- *functionality* – This goal focuses on changing the functional behavior of the source program. The semantic model for the source and target program are the same.
- *translation* – This goal focuses on transforming a source program into an equivalent target program having a different syntax and generally a different semantic model. Here both semantic models are roughly at the same level of abstraction.
- *computation* – This goal focuses on using transformations to perform computations. That is, one is interested in some form of evaluation of a program or expression.

3.1 Transformations that Shift Between Semantic Models

Compilation is a classic example of a fully automatic transformation whose source and target programs are understood with respect to different semantic models. The goal is *computability*. Source programs define computations that are typically understood in terms of semantic models containing high-level concepts such as variables, data structures, and recursion while target programs define computations that are understood in terms of semantic models consisting of registers, memory locations, bytes and bits.

Synthesis and *refinement* are two examples of activities in which source programs having specification-like characteristics are transformed into executable implementations. The goal is *computability*. Transformations in this realm are typically not fully automatic (otherwise they would be called compilers) and require some form of attention on a per problem basis. Specification languages can, and oftentimes do make use of constructs that are not computable. Thus, the semantic shift between source and target programs can be dramatic.

Migration is an activity in which a program written in one language is transformed into an equivalent program written in another language where both the source and target languages are roughly at the same level of abstraction (e.g., C++ and Java). The goal is *translation*. Such transformation can involve subtle shifts in semantic models. For example, the expression $(x++) + x$ has a precise semantics in Java and is technically undefined C++.

Aspect-oriented programming is a paradigm in which cross cutting aspects of software are defined separately [18][20]. These aspects are then *woven* together to form a program which can then be compiled and executed in a traditional fashion. The *weaving* of aspects into a program is typically approached from a transformation-oriented perspective. The goal in *weaving* is *translation*.

3.2 Transformations that Remain within a Single Semantic Model

In *partial evaluation* [16] knowledge that a general purpose source program will be used in a context where one or more of its inputs are fixed is used as the basis for transformation. The goal is *computability*. In particular, the target program produced is one in which all computations that can be performed statically have been carried out. Oftentimes this results in a dramatic improvement in the efficiency of the resulting program.

Desugaring is an activity where the goal of transformation is *simplification*. In desugaring the target program that is produced belongs to a language that is a strict subset of the language of the source program. The pseudo-compiler example given in Section 2.3 is an example of a desugaring transformation.

Renovation is an activity focusing on altering the behavior of a software system that is currently in use. The goal is *functionality*. The need for *renovation* is driven by changing requirements that are placed on the software.

Program *optimization* is a highly researched area in computer science. The goal in optimization is *efficiency*. Optimizations can occur at a variety of abstraction levels. A classic example can be found in [8] where an exponential algorithm for calculating fibonacci numbers is transformed into a linear time algorithm. Well-known optimizations include constant propagation, constant folding, strength reduction, and common sub-expressions elimination [2].

In the following sections we take a more in-depth look at two transformational activities that are, in some sense, at the opposite ends of the conceptual spectrum.

3.2.1 Refactoring

When developing software it is common to reach a point where some unanticipated structural or functional dependencies make the resulting software architecture difficult to understand and/or resistant to future modification. When such a point is reached programming effort needs to be expended to “clean up” the software. *Refactoring* is the term used to describe general techniques and methods that can be used to “clean up” software. More formally stated, the goal of refactoring is to restructure software to make it clearer (e.g., improving its design) while preserving its functionality. In contrast, the goal in *obfuscation* is to make software harder to understand.

Examples of refactoring range from simple to complex and include:

- *identifier renaming* – The goal of identifier renaming is to give a identifier a new name that more accurately describes its purpose.
- *method extraction* – The goal of method abstraction is to abstract a sequence of statements into a method.
- *object-oriented generalization* – The goal of generalization is to identify a collection of classes that share common features (e.g., methods and fields) and to migrate these common features to a super class.

- *object-oriented specialization* – The goal of object-oriented specialization is to identify a class containing a general abstraction whose realization consists of a number of distinct special cases. When such a class is discovered, a number of subclasses should be generated and each special case should be migrated into its own subclass.

Ideally, refactoring is accomplished by carrying out a sequence of small transformations each of which are so simple that they are “obviously” correctness-preserving. In addition to simplicity, these transformations also should build on one another in a cumulative fashion. Under these circumstances, a sequence of simple transformations can have an overall effect that results in a dramatic refactoring of the program. In many cases, refactoring is subjective activity. As a result, the ideal refactoring system is one that has an interactive dimension to it allowing users to actively participate in the refactoring process. Furthermore, such a system should support an undo operation that allows refactorings to be retracted, thereby allowing a variety of refactoring possibilities to be explored.

William Opdyke’s PhD thesis [23] is generally cited as the first major work that extensively looks at software refactoring as an area of research in its own right. However, in spite of this origination, the importance and implications of refactoring were not fully appreciated until popularized by Martin Fowler *et al* in a book titled *Refactoring – Improving the Design of Existing Code* [13]. Since then, software refactoring has become wide-spread. A number of tools are available to help software developers perform refactorings. Among these tools are: Transmogrify, Eclipse, RECODER, and RefactorIT. Refactoring has also been identified as an essential component of extreme programming [5].

3.2.2 The Evaluation of λ -expressions

Functional programming languages have their origins in a formalism known as the λ -calculus [4]. The syntax of the λ -calculus is extremely simple. The elements of the language of λ -calculus are called λ -expressions or expressions for short. A λ -expression can be a constant, a variable, the application of one λ -expression to another λ -expression, or a λ -abstraction of the form $(\lambda id.E)$ where *id* is an identifier and *E* is a λ -expression.

The λ -calculus is a powerful notation for describing general-purpose computation. In fact, it has been shown that any computable function can be described in terms of a λ -calculus expression. In this framework, computation consists of the evaluation of λ -expressions. The goal in an evaluation is to simplify a λ -expression until it can be simplified no further. If such a point is reached, we say the expression is in its *normal form*.

The manipulation of λ -expressions is governed by the three axioms shown below. The first two axioms make use of an operation that substitutes a value for all free occurrences of a variable within a λ -expression. Let *E* denote a λ -expression, let *x* a variable, and let *v* denote a value (i.e., a λ -expression). The expression $E[x \mapsto v]$ denotes the instance of *E* that is obtained by replacing all free occurrences of *x* in *E* by *v*. The first and third axioms make use of the ability to determine whether a variable *occurs free* within a λ -expression. The formal definitions of $E[x \mapsto v]$ and *occurs free* are straight-forward, but lie beyond the scope of this article. For more information see [4].

Axiom 1 *Alpha-conversion (variable renaming).* $\lambda x.E \xleftrightarrow{\alpha} \lambda y.E[x \mapsto y]$ provided *y* does not occur free in *E*.

Axiom 2 *Beta-conversion (function application).* $(\lambda x.E_1) E_2 \xleftrightarrow{\beta} E[x \mapsto E_2]$

Axiom 3 *Eta-conversion (redundant layers of λ -abstraction)*. $(\lambda x.F x) \xrightarrow[\eta]{} F$ provided x does not occur free in F and F is a λ -abstraction.

The equivalences in these axioms can be oriented from left to right to form corresponding *reductions* or rewrite rules. A λ -expression is simplified by applying reductions until the *normal form* of the expression is reached. When reducing λ -expressions, the workhorses of reduction are the rewrite rules derived from the second and third axioms, and a λ -expression to which these rules can be applied is called a *redex*.

An important corollary to a famous theorem known as the *Church-Rosser Theorem* states that normal forms for λ -expressions are unique (up to variable renaming). Given the knowledge of the uniqueness of normal forms, an interesting question to ask is: “Given a λ -expression E , can the normal form of E be reached by applying reductions in any order to any sub-expression of E ?” A second *Church-Rosser* theorem states that one is guaranteed to reach the normal form of a λ -expression (if it exists) by always reducing the leftmost outermost redex, and only applying α -conversion when needed to avoid the *name capture problem* (see [4] for more on the name capture problem).

4 Summary and Conclusion

In this article, *program transformation* is defined as a mechanism for manipulating programs (and other software artifacts) having its roots firmly grounded in *equational reasoning*. On an intuitive level, equational reasoning can be thought of as the notion that “equals can be replaced by equals”. Formalization of this notion makes use of concepts such as (1) *matching/unification*, (2) *confluence*, and (3) *termination*. The practical adaptation of the ideas underlying equational reasoning to the realm of meta-programming (i.e., program transformation) requires the use of parsing technology to automatically recognize the complex term structures that are typically possessed by computer programs. These term structures can be defined using context-free grammars and can be stored internally by the transformation system as (1) *parse trees*, which directly reflect the structure defined by the grammar, (2) *abstract syntax trees*, which capture the essence of the structure described by the context-free grammar, or even (3) *directed-acyclic graphs* (DAGS).

Applications lending themselves to a transformational perspective can be found in numerous areas including: compilation, refactoring, synthesis, refinement, and even computation.

Interest in program transformation is driven by the idea that, through their repeated application, a set of simple rewrite rules can affect a major change in a software artifact. From the perspective of dependability, the explicit nature of transformation exposes the software development process to various forms of analysis that would otherwise not be possible.

References

- [1] A. V. Aho, R. Sethi and J. D. Ullman. *Code Optimization and Finite Church-Rosser Systems*. Design and Optimization of Compilers (Ed. R. Rustin), Prentice-Hall, pp 89-106, 1972.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1988.
- [3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

- [4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised edition. Studies in Logic and the Foundations of Mathematics, volum 103, North-Holland, Amsterdam, 1984.
- [5] K. Beck. *eXtreme programming eXplained: Embrace Change*. Addison-Wesley, 1999.
- [6] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. *An Overview of ELAN*. In C. Kirchner and H. Kirchner, eds., International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science, France, 1998. Elsevier Science.
- [7] M. van den Brand, A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful*. IWPC 1998, June 24-26, Ischia, Italy.
- [8] R. M. Burstall and J. Darlington. *A Transformation System for Developing Recursive Programs*. Journal of the ACM (JACM), v.24 n.1, p.44-67, Jan. 1977.
- [9] R. D. Cameron. *An abstract pretty printer*. IEEE Softw., 5(6):61-67, Nov. 1988.
- [10] H. Cirstea and C. Kirchner. *Intoduction to the rewriting calculus*. INRIA Research Report RR-3818, December 1999.
- [11] G. Dowek. *Higher-order Unification and Matching*. Handbook of Automated Reasoning, Volume 2, pp 1009-1062, 2001.
- [12] S. Eker. *Associative-commutative matching via bipartite graph matching*. Computer Journal, 38(5):381-399, 1995.
- [13] M. Fowler, K. Beck, J. Bryant, W. Opdyke, and D. Roberts. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [14] HATS. <http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html>
- [15] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction fo Automata Theory, Languages, and Computation (second edition)*. Addison Wesley, 2001.
- [16] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [17] D. Kapur and P. Narendran. *Double-exponential Complexity of Computing a Complete Set of AC-Unifiers*. Logic in Computer Science (LICS), Santa Cruz, CA, June 1992.
- [18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. Springer-Verlag, LNCS 1241, 1997.
- [19] R. Lämmel. *Typed Generic Traversal With Term Rewriting Strategies*. Journal of Logic and Algebraic Programming, Vol 54, pp 1-64, 2003.
- [20] C. V. Lopes and G. Kiczales. *D: A Language Framework for Distributed Programming*. Technical report SPL9710047 Xerox Palo Alto Research Center, February 1997.
- [21] A. Martelli and U. Montanari. *An efficient unification algorithm*. ACM Trans. Programming Lnaguages and Systems, 4(2):258-282, 1982.
- [22] T. Mens and T. Tourw. *A Survey of Software Refactoring*. IEEE Transactions on Software Engineering, Volume 30, Number 2, pp. 126-139, February 2004
- [23] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD Thesis, University of Illinois at Urbana-Champaign.
- [24] L. F. Rubin. *Syntax-directed pretty printing a first step towards a syntax-directed editor*. IEEE Trans. on Softw. Eng., SE-9(2):119-27, Mar. 1983.

- [25] R. Stansifer. *The Study of Programming Languages*. Prentice Hall, 1995.
- [26] M. Tomita. *Efficient Parsing for Natural Languages – A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.
- [27] E. Visser. *A survey of rewriting strategies in program transformation systems*. In B. Gramlich and S. Lucas, editors, Workshop on Reduction Strategies in Rewriting and Programming (WRS'01), volume 57 of Electronic Notes in Theoretical Computer Science, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
- [28] E. Visser. *Scoped dynamic rewrite rules*. In M. van den Brand and R. Verma, editors, Rule Based Programming (RULE'01), volume 59/4 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, September 2001.
- [29] E. Visser, Z. Benaissa, and A. Tolmach. *Building Program Optimizers with Rewriting Strategies*. Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98).
- [30] V.L. Winter and M. Subramaniam. *The Transient Combinator, Higher-Order Strategies, and the Distributed Data Problem*. Science of Computer Programming.
- [31] V.L. Winter, S. Roach, G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software*. Advances in Computers vol 58.
- [32] V.L. Winter. *Strategy Construction in the Higher-Order Framework of TL*. The 5th International Workshop on Rule-Based Programming (RULE 2004).