How Understanding and Restructuring differ from Compiling —a Rewriting Perspective—

Paul Klint

Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 AB Amsterdam, The Netherlands
`http://www.cwi.nl/~paulk`

Abstract

Syntactic and semantic analysis are established topics in the area of compiler construction. Their application to the understanding and restructuring of large software systems reveals, however, that they have various shortcomings that need to be addressed. In this paper, we study these shortcomings and propose several solutions. First, grammar recovery and grammar composition are discussed as well as the symbiosis of lexical syntax and context-free syntax. Next, it is shown how a relational calculus can be defined by way of term rewriting and how a fusion of term rewriting and this relational calculus can be obtained to provide semantics-directed querying and restructuring. Finally, we discuss how the distance between concrete syntax and abstract syntax can be minimized for the benefit of restructuring. In particular, we pay attention to origin tracking, a systematic technique to maintain a mapping between the output and the input of the rewriting process. Along the way, opportunities for further research will be indicated.

1. Introduction

Any text book on compiler construction (e.g., [2, 45]) tells us that a compiler consists of three phases: (*i*) syntax analysis; (*ii*) semantic analysis; and (*iii*) code generation. Today, most compiler research has shifted to just-in-time compilation, profile-driven code optimization and energy-aware code generation. Although there are still open issues in areas like inter-procedural pointer analysis and aliasing, and code generation for specialized chip sets and parallel architectures, it is fair to say that many aspects of compilation can be considered a closed topic. This is particularly true for syntax analysis and major parts of semantic analysis. Of course, this does not imply that all these techniques are already applied in practice.

Since the understanding and restructuring of software systems are tasks that resemble compilation, it is a widely held belief that the understanding and restructuring of software systems can be achieved by simply applying standard compilation techniques. In this paper I will discuss the mismatch between standard compiler techniques and the requirements imposed by software system understanding and restructuring. Semantic issues are approached from a rewriting perspective and opportunities for further research will be indicated along the way.

1.1. Compilation

Compilation is a well-defined process for the construction of new software with well-defined input, output and constraints. This can be detailed as follows:

- Input is a source program in a fixed language with a fixed syntax definition and semantics.
- Output is code in a fixed target machine language with well-defined syntax definition and semantics.
- The constraints imposed on the compilation process are known in advance. First and foremost, the generated machine code should be semantically equivalent to the source program. In addition to this, certain parameters may steer the compilation process, e.g., efficiency of the compilation process itself, or performance parameters of the generated code (efficiency, memory footprint, power consumption, etc.).
- Compilation is a batch-like process without human intervention.
- Compilers are used for *forward* engineering, i.e., the construction of new software.

1.2. Understanding and Restructuring

System understanding is an exploration process that may have two forms of input: (*i*) all explicit artifacts related to a software system such as, for example, design documents,

source code, revision histories, build files, test cases, and documentation; (*ii*) implicit knowledge about the system that its designers, builders or maintainers may still have. There is no clear target language for system understanding. The extracted facts can be produced as a textual listing, as database records or as a graphical display.

System understanding is an interactive, iterative, process that follows the Extract-Abstract-View scenario:

Extract: Parse source text and extract elementary facts Note that the extraction phase depends on the source language.

Abstract: Perform operations on these elementary facts to obtain derived facts that are tailored towards the analysis problem at hand.

View: Use the derived facts for textual reporting or visualization.

System restructuring is a transformation process that takes the source code (and possibly other artifacts) of a given software system as input and generates new source code that is "better" according to given criteria. Improvements may amount to removal of obsolete language constructs, restructuring of the control flow, or even complete regrouping and rearranging of code in order to get a better overall system architecture. The resulting code may be in the same language (source-to-source transformation) or in another language (language conversion).

The goals of understanding and restructuring are more diffuse and varied than is the case for compilation. There is no *a priori* target for representing basic facts, and the desired queries and inferences may vary widely. Understanding and restructuring are applied to existing software. They have therefore the following properties that differentiate them from compiling:

- Systems consist of several sources languages; for some of these no proper syntax definition is available and this may have to be recovered. Semantics maybe unclear.
- Analysis has to be performed over these multiple source languages. This implies a need for language-independent analysis frameworks.

  Not only traditional compiled languages like COBOL, PL/I, FORTRAN, C, C++ and Java have to be considered, but also scripting language like Tcl, Perl, PHP and others that are largely interpreter-based and defeat many techniques for static analysis.
- A compiler abstracts away from the source text as soon as possible. Understanding and restructuring require closer links to the source text due to textual/syntactic queries and the need to literally reconstruct the source during restructuring. This introduces additional requirements such as the traceability from derived facts or restructured code to the source code.
- The process of system understanding is highly interactive: initial findings may trigger new questions that lead to new findings, and so on. In addition to this, the results of understanding may be viewed and browsed in various manners.
- Understanding and restructuring are used for *reverse* engineering, i.e., the understanding and improvement of existing software.

1.3. Why Rewriting?

Assuming some form of lexical or syntactic analysis (we come back to this in Section 2), there are two main options available for further analysis and restructuring.

The first option is to use a standard programming language (C, C++, Java, C#) to explicitly program the desired analysis and restructuring. Libraries may be available to support these tasks. This is the dominant approach today, but is has several drawbacks. It is not so easy to express the pattern matching that is required to extract information and to identify the starting points for restructuring. This code is hard to maintain and not easy to modify when the requirements change.

The second option is to use some higher-level, rule-based, formalism to express analysis and restructuring. The major contenders here are attribute grammars and term rewriting systems. Advantages are that pattern matching can be expressed at a high level and that many high-level features are provided: traversal primitives, construction primitives, etc.

The focus in this paper is on *term rewriting systems*. The simplification of algebraic expressions or formal differentiation form a good basis for understanding them. Rewrite rules consist of a left-hand side (a pattern to be matched) and a right-hand side (the replacement term). Term rewriting amounts to taking an initial term T_0 and applying the rules as long as possible. This leads to a number of intermediate steps $T_1, ..., T_{n-1}$ and when no more replacements are possible the answer (in jargon: the normal form) T_n is found.

Term rewriting systems are simpler than attribute grammars and give more possibilities for mixing computations on syntax trees and on semantic values. Term rewriting scales well to huge applications. The ASF+SDF Meta-Environment [26, 5, 16, 8, 7] serves as a conceptual background and long-term perspective for the topics discussed in this paper, but we do not assume any knowledge of this technology and will explain details as needed. Many of

the considerations also apply to other systems supporting term rewriting. There are, in particular, many commonalities with TXL [13]. We refer to [21] for a reasonably complete overview of term rewriting systems.

1.4. Plan for this Paper

First, we will address topics related to syntactic analysis in Section 2: the recovery of lost grammars (Section 2.1), the need to compose grammars from grammar modules (Section 2.2), and the reconciliation of lexical syntax with context-free syntax (Section 2.3).

Next, we move on to semantic issues in Section 3: after a general discussion on dataflow analysis (Section 3.1) we focus on the use of relations for representing program facts and querying them. In Section 3.2 we develop a relational calculus for this purpose and in Section 3.3 we sketch a perspective in which term rewriting and relational calculus are fused into a powerful tool for both syntactic and semantics queries and transformations.

In Section 4 we discuss restructuring and focus on techniques that minimize the distance between the source of a restructuring and its result. We discuss the need to bring concrete syntax and abstract syntax closer together (Section 4.1) and the need to preserve annotations during rewriting (Section 4.2). In Section 4.3 we describe origin tracking, a systematic technique to maintain a mapping between the output and the input of the rewriting process. Section 5 gives some closing remarks.

2. Syntax Analysis

Context-free languages, parsing (syntax analysis) and parser generation have been widely studied in the middle of the previous century. A standard work summarizing the field in the early seventies is [3]. It is not surprising that many consider this topic as closed and that it is not so easy these days to get a paper on this subject accepted for publication. These results are used today in many parsers and parser generators.

There are, however, reasons to reconsider this subject from the perspective of understanding and restructuring.

2.1. Grammar Recovery

The basic assumption of all work on context-free languages is that a syntax definition is a given artifact that can be analyzed and used. Analysis amounts to defining subclasses of context-free languages with certain properties (e.g., unambiguous, finitely ambiguous, inherently ambiguous, deterministically parseable) and to search for criteria to test whether a given language or syntax definition falls in some of these subclasses. Use of a syntax definition amounts to the generation of parsers for some of these subclasses.

A topic that is never addressed in this work is how syntax definitions come into existence. In fact, a complete engineering process can be identified that deals with the design, construction, testing, use, improvement and maintenance of syntax definitions. Some initial results on this subject are reported in [29, 14]. Major questions to be addressed are:

- The extraction of syntax definitions from manuals, tools and online resources.

- The systematic transformation and improvement of these extracted syntax definitions. This includes, amongst others, the disambiguation of the type information that is encoded in names of nonterminals (e.g., `<IntegerExpression>`). It also has to bridge the gap between the original purpose a grammar was written for (e.g., standardization or documentation) and the way we want to use it (e.g., tool generation).

2.2. Grammar Composition

Another topic that is only scarcely addressed in the work on context-free languages is the issue of *modularity* of syntax definitions. By this we mean the decomposition of a syntax definition in a number of modules (e.g., separate modules containing syntax rules related to procedure declarations, record definitions, database operations, or exception handling.) and the composition of such modules to form a complete syntax definition. The merit of this approach is that a clear separation can be made between a base language (e.g., COBOL) and its dialects or extensions (e.g., COBOL VS II, extensions for CICS or SQL). A desired combination of dialect features and extensions can be readily assembled from syntax modules.

Unfortunately, the widely available parser generators only accept syntax definitions that fall in subclasses of the context-free grammars (LL(1), LALR(1), etc.) that are not closed under composition. For instance, a syntax definition for yacc should be LALR(1) and the combination of two LALR(1) definitions is not guaranteed to be again LALR(1). This becomes clear during parser generation time when so-called shift/reduce conflicts or reduce/reduce conflicts are discovered. Both indicate that the parser generator cannot make a unique choice how to generate a parser that behaves correctly under all circumstances.

One approach to solve this problem is to use *generalized parsing*[30, 41, 22], a technique that takes the existence of conflicts for granted and tries to resolve them during parse time (as opposed to parser generation time when they are discovered). The net effect is that the complete class of context-free languages can be parsed. The price to

be paid is that the parser has to pursue more than one possible parse simultaneously while parsing according to certain syntax rules. Some of these parallel parses will lead to a complete parse, others will be aborted half way since they do not encounter the required input. This method has been validated in practice but research questions remain:

- Since the complete class of context-free languages can be recognized, also ambiguous sentences can be recognized. The filtering of undesired ambiguities is an area of ongoing research [11].
- In the case of a syntactically incorrect input sentence, more than one parse may have been tried. This makes it hard to give precise error messages pointing at the precise location of and reason for the error.
- The parameterization of modular grammars needs further study.

2.3. Lexical versus Context-free Grammars

In the context of program understanding, there are two major options for the textual analysis of programs: lexical analysis and context-free analysis.

Using *lexical analysis* boils down to using regular expressions as the basis for analysis and using tools (e.g., sed, grep, awk, lex) or languages (e.g., Perl, Python, Java) with support for them. The major advantage of this approach is that only regular expressions are needed for the language constructs we want to analyze (e.g., call statements) and that the availability of a full syntax definition of the language in question is not required and thus circumventing the grammar recovery problem discussed above (Section 2.1). Another advantage is that even syntactically incorrect or incomplete programs can be analyzed. The disadvantage of this approach is that it becomes harder and harder to maintain and control the set of regular expressions as the number of language constructs under investigation increases.

Using *context-free analysis* amounts to writing a context-free syntax definition for the language(s) of interest and using a parser generator to generate a parser. The use of a complete context-free grammar has as advantage that one gets a coherent overview of all language constructs. The disadvantage is that the creation of such a grammar may be a substantial effort (see again Section 2.1).

An interesting intermediate approach are *island grammars* [33]. The basic idea is to distinguish between *islands* (relevant constructs which are parsed in detail) and *water* (irrelevant constructs which are skipped). Islands grammars can be implemented in a natural manner using generalized parsing as discussed above (Section 2.2) and scanner-less parsing ([36, 43]). The former is necessary since islands do not fall easily in one of the standard subclasses of context-free languages. The latter facilitates the seamless integration of lexical syntax and context-free syntax and is necessary to achieve character-level parsing as is achieved with regular expression based scanning.

In scanner-less parsing there is (surprise) no scanner and only a parser: each individual character is considered as a lexical token and all scanning decisions are made by the parser.

The notion of island grammars is appealing but more experience is needed to determine their merits. Research questions are:

- The design of island grammars is not easy and requires a lot of experimentation. Is it possible to develop a systematic design method for island grammars?
- How can island grammars be validated, i.e., is it possible to determine that the island grammar recognizes all relevant constructs?
- Is it possible to design *modular* island grammars? Currently, the definition of water in different definitions easily interferes with the definition of islands in another grammar. See also the discussion in Section 2.2.

3. Semantic Analysis

Semantic analysis is a major subject in both compiler construction and system understanding and addresses topics like dataflow analysis, pointer and alias analysis, type inferencing, slicing, run-time profiling, coverage analysis, and much more.

We have seen above that the analysis of multiple languages is a big issue for system understanding and we will approach semantic analysis from this perspective and take dataflow analysis as an example.

3.1. Dataflow Analysis

Dataflow analysis is one of the techniques used to collect information about the use, definition and dependencies of data in programs. In [20] a generic dataflow framework is developed that can be instantiated to solve problems like reaching definitions, live variables, constant folding, dead code elimination, and others. From the perspective of system understanding (and its need to handle multiple languages) it would make sense to use tools that implement such a framework and instantiate them for the required languages and dataflow problems. Several attempts have been made to build such generic dataflow tools [25, 32, 4] but they have never come into wide-spread use. In the case of compiler construction, dataflow analysis is implemented from scratch for each new compiler. There are, of course, several

compiler frameworks that provide dataflow tools or libraries with routines to support dataflow analysis, but the point is that it is either not easy to adapt these tools to other languages than supported by the framework, or they are hard to use in isolation or in combination with tools outside the framework.

We can only speculate about the reasons for this state of affairs. It may be that there is a mismatch between the functionality provided by the generic tools and the needs of potential applications. Recall, in the case of understanding and restructuring, the need for analyzing multiple languages and the wide range of queries. It is also possible that the savings by generic dataflow tools do not compensate the additional effort of learning and using such tools.

3.2. Relations

Algorithms for dataflow analysis are usually presented as *graph* algorithms and this seems to be at odds with our emphasis on *term* rewriting, the major difference being that graphs can and terms cannot contain cycles. Fortunately, every graph can be represented as a relation and it is therefore natural to have a look at the combination of relations and term rewriting.

The idea to represent relational views of programs is already quite old. For instance, in [31] all syntactic as well as semantic aspects of a program were represented by relations and SQL was used to query them. Due to the lack of expressiveness of SQL (notably the lack of transitive closures) and the performance problems encountered, this approach has not seen wider use. In Rigi [34], a tuple format (RSF) is introduced to represent relations and a language (RCL) to manipulate them. In [35] a *source code algebra* is described that can be used to express relational queries on source text. In [10] a *query algebra* is formulated to express direct queries on the syntax tree. It also allows the querying of information that is attached to the syntax tree via annotations. Relational algebra is used in GROK [24] and Relation Partition Algebra (RPA) [19] to represent basic facts about software systems and to query them. In GUPRO [18] graphs are used to represent programs and to query them. In F(p)–ℓ [12] a Prolog database and a special-purpose language are used to represent and query program facts.

3.2.1. Relations and Term Rewriting

How can relations and term rewriting help answering queries about syntactic and semantic aspects of software systems? Due to the mathematical nature of relational algebra it comes as no surprise that bags, relations and the operations on them can be defined easily using term rewriting. For instance, the difference of two bags `Bag1 \ Bag2` is defined by:

```
[df-1] {Es1, E, Es2} \ {Es3, E, Es4} =
       {Es1, Es2} \ {Es3, Es4}
[default-df-2]
       {Es1} \ {Es2} = {Es1}
```

We use the ASF+SDF notation, but this can be expressed easily in any term rewriting formalism. Points to consider are the following. We use the traditional notation for Bags: lists of elements enclosed by `{` and `}`. We use the convention that variables `E`, `E'`, `E1`, `E2`, ... stand for individual elements and that the variables `Es`, `Es1`, `Es2`, ... stand for lists of zero or more elements. In `[df-1]` the case is handled that `Bag1` and `Bag2` still have an element `E` in common that has to be removed. Although `Bag1` and `Bag2` may have more than one element in common, we carefully remove one common element at a time in order to get the correct number of repetitions of elements in the resulting bag. In `[default-df-2]` the complementary case is handled that `Bag1` and `Bag2` have no element in common.

In a similar spirit operations on relations can be defined. For instance, the range operator `ran` that takes a relation (a bag of tuples) and projects each tuple to its second element:

```
[ran-1]  ran( {<E,E'>, Tuples} ) =
         {E'} union ran( {Tuples} )
[ran-2]  ran( {} ) = {}
```

The variable `Tuples` stands for a list of zero or more tuples. Along these lines it is straightforward to define a complete menagerie of operators on bags and relations. An interesting issue is that we have not yet committed to a particular kind of *elements* for these bags and relations. By modeling this as a formal parameter of both datatypes we can select the kind of elements as needed. In this way we can even include (fragments of) syntax trees in relations.

3.2.2. A Relational Calculus

In relational *algebra*, queries are expressed by applying specialized operators to relations. In relational *calculus*, queries describe a desired set of tuples by specifying the predicate the tuples must satisfy. See [42] for a complete coverage of this distinction.

As an experiment, we define a little relational calculus (let's call it RSCRIPT) with the usual operators on bags and sets such as `union`, `elem` (is element of), `==` (equality) and more. We also use bag and relation formers that resemble list comprehensions in functional languages [44]. The main difference is that we make a stricter distinction between value generators, predicates and constructing expression as we explain now.

Let a *generator* be either (*i*) a variable ranging over the elements of a bag: `Var in Exp`, where `Exp` is a bag-valued expression; or (*ii*) a tuple of two variables ranging over the elements of a relation: `<Var1, Var2> in`

`Exp`, where `Exp` is a relation-valued expression. We can now define two kinds of bag or relation formers:

- { `Gen | Pred` }: results in a new bag or relation that contains all elements in the generator `Gen` that satisfy the Boolean-valued expression `Pred`.
- { `Gen1, ... | Pred | Exp` }: results in a new bag or set that is obtained by considering all combinations of elements from the various generators, determining which combinations satisfy `Pred`, and for those construct a new element or tuple `Exp` for the resulting bag or relation.

Observe that most traditional primitives from relational algebra can be defined using the bag and relation formers. For instance, the `ran` function we have seen earlier can be defined as:

```
fun ran(R) = { <X, Y> in R | true | Y }
```

In a similar fashion, the inverse `inv` and the composition `o` can be defined:

```
fun inv(R) = { <X, Y> in R | true | <Y, X> }

fun o(R1, R2) =
    { <X, Y> in R1, <S, T> in R2
    | Y == S | <X, T>}
```

Essential operators that cannot be defined in this manner are (variants of) the transitive closure operator. We provide the usual transitive closure operator R^+ defined by $R^+ = \bigcup_{i=1}^{N} R^i$. Here N is the cardinality of R and R^i denotes i compositions of the relation R: $R \circ R \circ ... \circ R$. We also define $R^0 = Id$ where Id is the identity relation. The reflexive transitive closure R^* is defined by $R^* = Id \cup R^+$.

As an experiment, we also provide built-in functions for computing more sophisticated closures. For instance, if U is a relation representing uses of variables, D is a relation representing definitions of variables, and P is the backward control flow relation, then $dominators(U,D,P)$ computes which nodes can reach the use nodes without visiting a definition node. This function is very similar to the *dominates* relation used in dataflow analysis [2]. We define it by first introducing the auxiliary function d:

$$\begin{aligned} d^0(U,D,P) &= U \\ d^n(U,D,P) &= (d^{n-1}(U,D,P) \circ P) \setminus D \end{aligned}$$

In this definition, d^0 has U as initial value, and the subsequent d^i represent the next composition with the relation P, thus extending the closure one step further. In each step, however, elements from D are left out, so the closure operation stops for those elements. Now we can define

$$dominators(U,D,P) = \bigcup_{i=0}^{N} d^i(U,D,P)$$

where N is the cardinality of the relation P. Observe that $R^+ = dominators(R,\{\},R)$, so *dominators* is really a generalization of R^+. There are other ways to generalize the transitive closure as described in [1].

This completes the definition of *dominators*; an application follows shortly. Experience has to show which generalized closure operations we really need. This also concludes the sketch of RSCRIPT. The points to stress are:

- Bags and relations can easily be defined using rewrite rules.
- The type of elements in bags and relations can be a parameter of these datatypes; in this way also syntactic information can be included in relations.
- An interpreter for RSCRIPT can also easily be expressed with rewrite rules (we do not discuss it here).

3.2.3. A Toy Example

To make the above sketch of RSCRIPT more tangible consider the following toy program:

```
declare x, y, z : integer;
 [n1] x := 3;
      if [n2] 3 then
         [n3] z := y + x
      else
         [n4] x := 4
      fi
 [n5] y := z
```

It consists of the declaration of three integer variables `x`, `y` and `z` followed by three statements. Points of interest are labeled with `[n1]`, `[n2]`, and so on. It is straightforward to extract the relations `DEFS` (definitions of variables), `USES` (uses of variables) and `SUCC` (the successor relation that represents the control flow graph, including the start node `n0`):

```
def DEFS = '{<x, n1>, <z, n3>, <x, n4>,
             <y, n5>}
def USES = '{<y, n3>, <x, n3>, <z, n5>}
def SUCC = '{<n0, n1>, <n1, n2>, <n2, n3>,
             <n2, n4>, <n3, n5>, <n4, n5>}
```

We use a single quote (`'`) to denote literal bags or sets. Uninitialized variables can now be found as follows:

```
def PRED  = inv(SUCC)
fun rr(R) = elem('n0, ran(R))
def UNDEF =
  { <V, N> in USES |
    rr(dominators({<V,N>}, DEFS, PRED)) }
```

First, we determine the inverted control flow relation `PRED`. Next, we define an auxiliary function `rr` (for "reaches root") that checks whether the start node `n0` is in the range of a given relation `R`. Finally, `UNDEF` determines for each tuple in the `USES` relation, whether it can be reached from

the start node `n0`. This is done by way of the `dominators` function explained earlier. If so, it represents a possible uninitialized usage of the variable. In this example, the result will be {`<z,n5>, <y,n3>`} and this is as expected: the use of variable `y` at `[n3]` is not preceeded by *any* initialization. For the use of variable `z` at `[n5]` the problem is that not *all* preceeding execution path contain a definition.

3.3. Perspective and Research Issues

Relations form a language-independent representation of facts extracted from programs and this approach is well-suited for the analysis of multiple-language systems. The original source text contains all the details about primary syntactic aspects of the programs and the relations contain secondary extracted information.

An intriguing perspective starts to emerge here in which term rewriting and relational calculus can be fused. In the described set-up nothing prevents us from combining the derived relations with rewrite rules that perform either syntactic queries or restructuring of the original source program. This makes it possible to perform queries that use both syntactic information and derived semantic information, e.g., "give all procedure calls that can modify variable X". This makes it also possible to perform restructuring based on both syntactic and semantic information.

In addition to the Extract-Abstract-View scenario sketched earlier (Section 1.2), we get two further scenarios: Extract-Abstract-Semantic Queries-View and Extract-Abstract-Semantic Transformations. The two new steps are

Semantic Queries: Perform queries on the syntax tree using derived semantic information obtained in the abstraction step. This allows, for instance, to formulate syntactic queries that also need dataflow information.

Semantic Transformations: Transform the original syntax tree using derived semantic information obtained in the abstraction step. This allows the formulation of transformations that need semantic information.

The contribution here is to create a simple, rewriting based, language-independent framework that supports both analysis and transformation using syntactic and semantic information. The latter can be obtained using relational calculus. However, many questions remain:

- The evaluation of the relation-based approach versus the graph-based approach is a matter of ongoing debate [23]. There are at least two distinctions to be made: algebra *vs.* calculus and relations *vs.* graphs. Our (current) position in this matter is that a calculus that includes certain reachability primitives is simpler for the user and that relations *vs.* graphs is a matter of implementation. Clearly more research and evaluation are needed.
- Although the algorithms to implement relational calculus are mostly known, the integration with rewriting poses an implementation challenge.
- How to deal with the need for incremental updates of the derived relations after a restructuring step?

4. Restructuring

Restructuring and translation are the areas where term rewriting really shines. Nothing is more natural than to say: "replace construct A by construct B provided that conditions $C_1, ..., C_n$ are satisfied". This is precisely what a conditional rewrite rule is about. We will neither explain this in detail (but see [5, 16, 38]) nor elaborate on the successes of term rewriting in this area (but see [8]). Instead, we will focus on three very specific topics that manifest themselves when applying term rewriting to understanding and restructuring. All three topics aim at minimizing the distance between the source of a restructuring and its result.

4.1. Concrete versus Abstract Syntax

In compiler construction it is usual to abstract from the source text as soon a possible: from source text to parse tree to abstract syntax tree to some form of intermediate representation that is suitable for optimization and code generation.

Since restructuring frequently deals with source-to-source transformations in which the result has to stay as close as possible to the original, it is important to minimize the distance between the rewriting rules and the concrete syntax of the language. It is also well-known that for a realistic language the abstract syntax is defined by hundreds of operators and that the writing of transformation rules becomes harder and harder.

The solution as adopted in ASF+SDF is to allow arbitrary concrete syntax in the rewrite rules. Consider as a trivial case, the `and` function in Boolean expressions. Rather than using the prefix notation `and(B,false)` one can also allow the infix notation `B & false`, where `B` is a Boolean variable. Instead of the rewrite rule

```
[and-1] and(B, false) = false
```

one can now write

```
[and-2] B & false = false
```

This seems nice, but not spectacular. The benefits become, however, clear if we consider the effect on restructuring rules for programming languages: we can write the rules using the usual concrete syntax for the constructs to be transformed with variables occurring at appropriate places. This

simplifies the understanding of these rules. For instance, the following rule comes from a commercial system that adds, amongst many other things, `END-IF` keywords to if statements in COBOL:

```
[ai-1] addEndIf(IF Expr OptThen Stats)
       = IF Expr OptThen Stats END-IF
```

Concrete syntax in rewrite rules has many advantages, but it may occasionally hinder abstraction. The best approach is probably to limit the use of concrete syntax to those applications where the above advantages can be obtained. See [37, 14] for a further discussion of this topic.

4.2. Preserving Annotations

A second topic to be discussed is the role of annotations in the rewriting process. Given a set of rewrite rules, rewriting is based on the idea that some initial term is traversed and that wherever a left-hand side of a rewrite rule matches, it is replaced by the corresponding right-hand side. Now it happens frequently that tools want to transparently add additional information to a term for later usage. Examples are coordinates in the source code, the focus of an editor, pointer to documentation node, etc.

Two steps are needed to achieve rewriting with annotations. First, the term representation has to be extended with annotations [9]. Second, the rewriting process has to be extended to transparently handle annotations. The idea of rewriting with annotation was first proposed in [28] where annotations are called labels. Application of rewriting with annotations to the origin tracking problem (to be discussed shortly in Section 4.3) were proposed independently in [27] and [6]. The latest version of the ASF+SDF Meta-Environment supports rewriting with annotations.

4.3. Origin Tracking

Compilers are only concerned with the original source text when they have to produce warnings or error messages: coordinates in the original source text and a message that applies to the program text at that location. Computing correct coordinates is not easy, in particular when highly optimized code has been generated. Usually, ad hoc techniques are used to maintain these coordinates, but see [39].

In system understanding and restructuring the need for good coordinates in the source text is evident when answering questions like: "where does this variable occur in the text?" or "what is the original of this restructured code?".

In [17] we have proposed a technique called *origin tracking* to maintain such dependencies in a completely general way. It crucially depends on term rewriting. Suppose that the ordinary term rewriting process proceeds in the steps $T_0 \rightarrow T_1 \rightarrow ... \rightarrow T_n$. Here T_0 is the initial term (probably a function like an interpreter or compiler applied to the syntax tree of a program) and T_n is the normal form (the value computed by the program or the code generated by the compiler). The key idea of origin tracking is to compute a reverse mapping $\mapsto$ between the steps in the derivation: $T_n \mapsto T_{n-1} \mapsto ... \mapsto T_0$. Various methods can be used to compute this information. For instance, one can annotate T_0 with source text coordinates and apply rewriting with annotations. The annotations that are preserved in T_n then point to origins in T_0. For instance, given rewrite rules for the reversal of a list, `reverse([one,two,two])` will yield in a few steps `[two,two,one]`. The question now arises where each `two` comes from. We answer this in two steps. First annotate the initial term; this leads to `reverse([a:one,b:two,c:two])`. The next step is to rewrite it with annotations and this yields `[c:two,b:two,a:one]`. We can now immediately answer the question about the `two`s.

This approach is ideal when the normal form T_n still contains fragments of the original term T_0. This is typically the case for fact extraction, static analysis and source-to-source transformation. In [40] it is shown how this works for error reporting. In the case of language conversion (translation to another language) this information may be insufficient and additional origin relations have to be established, but see [15]. Research questions are:

- It is unlikely that a fixed set of propagation rules for origin information can be defined that are suitable for all applications. A more "programmable" method is to be preferred.
- The algorithms needed for the efficient calculation of origin relations are largely unexplored.
- The relation between origin tracking and traversal strategies for terms needs further study.

5. Concluding Remarks

Understanding and restructuring of software systems impose other requirements on syntactic and semantic analysis than compiling. In this paper we have only briefly discussed syntactic issues: recovery and composition of grammars and integration of lexical and context-free syntax. For semantic issues we have used term rewriting: a simple, unifying, framework that is very well understood from a theoretical perspective and scales well to industrial applications. We have briefly explored the fusion of term rewriting and relational calculus as well as techniques to minimize the distance between the source of a restructuring and its result. Although an in-depth treatment of these topics is out of the scope of this paper, we hope that this overview gives a useful impression of the ongoing research efforts in this area and acts as an invitation to join them.

Acknowledgments

The comments made by Arie van Deursen, Leon Moonen and Jurgen Vinju are greatly appreciated.

References

[1] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, July 1988.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Englewood Cliffs (NJ), 1972–73. Vol. I. Parsing. Vol II. Compiling.

[4] M. Alt, F. Martin, and R. Wilhelm. Generating Dataflow Analyzers with PAG. Technical Report A10-95, Universität des Saarlandes, 1995.

[5] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[6] I. Bethke, J.W. Klop, and R.C. de Vrijer. Descendants and origins in term rewriting. *Information and Computation*, 159(1-2):59–124, 2000.

[7] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[8] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[9] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[10] M. G. J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *Lecture Notes in Computer Science*, pages 235–255. Springer-Verlag, 1996.

[11] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[12] G. Canfora, A. de Lucia, and G.A. Lucca. A system for generating reverse engineering tools: a case study of software modularisation. *Automated Software Engineering*, 6:233–263, 1999.

[13] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[14] T.R. Dean, J.R. Cordy, A.J. Malton, and K.A. Schneider. Grammar programming in TXL. In *2nd International Workshop on Source Code Analysis and Manipulation (SCAM02)*. IEEE, October 2002.

[15] A. van Deursen. *Executable Language Definitions - Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.

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

[17] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.

[18] J. Ebert, R. Gimnich, H.H. Stasch, and A. Winter. *GUPRO: Generische Umgebung zum Programverstehen*. Koblenzer Schriften zur Informatik. Fölbach, 1998.

[19] L.M.G. Feijs, R. Krikhaar, and R.C. Ommering. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, april 1998.

[20] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, Amsterdam, 1977.

[21] J. Heering and P. Klint. Rewriting based languages and systems. In Terese, editor, *Term Rewriting Systems*, chapter 15. Cambridge University Press, 2003.

[22] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *SIGPLAN Notices*, 24(7):179–191, 1989.

[23] R. C. Holt, A. Winter, and J. Wu. Towards a common query language for reverse engineering. Technical report, Institute for Computer Science, Universität Koblenz-Landau, August 2002.

[24] R.C. Holt. Binary relational algebra applied to software architecture. CSRI 345, University of Toronto, march 1996.

[25] G. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[26] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[27] P. Klint. From line numbers to origins. In E.H.L Aarts, H.M.M. ten Eikelder, C. Hemerik, and M. Rem, editors, *Simplex Sigillum Veri: Liber Amicorum for Prof.dr. F.E.J. Kruseman Aretz*, pages 215–230. Technical University Eindhoven, 1995.

[28] J.W. Klop. *Combinatory Reduction Systems*. Number 127 in Mathematical Centre Tracts. CWI, 1980. PhD Thesis.

[29] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[30] B. Lang. Deterministic techniques for efficient nondeterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[31] M. A. Linton. Implementing relational views of programs. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 132–140, 1984.

[32] L. Moonen. A generic architecture for data flow analysis to support reverse engineering. In M.P.A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Amsterdam, November 1997. Springer-Verlag.

[33] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, October 2001.

[34] H. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE 10),*, pages 80–86,, April 1988.

[35] S. Paul and A. Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

[36] D.J. Salomon and G.V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178, 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[37] M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society, 1998.

[38] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[39] C. Tice and S.L. Graham. OPTVIEW: a new approach for examining optimized code. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 19 – 26, 1998.

[40] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(1):5–55, 2001.

[41] M. Tomita. *Efficient Parsing for Natural Languages—A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.

[42] J.D. Ullman. *Principles of Database Systems, Second Edition*. Computer Science Press, 1982.

[43] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[44] P. L. Wadler. List comprehensions. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 15. Prentice Hall, 1987.

[45] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1996.