Abstract

We characterise a computational model for processing annotated parse trees. The model is basically rewriting-based, i.e., the basic units of functionality for processing parse trees are rewrite rules that perform tree matching and tree building. The original ingredients of the model are provisions for dealing with annotations along the ordinary rewrite steps. There are progression methods, which define generically how to compute the annotations of the output from the annotations of the input. There are also access methods, which can be used in the rewrite rules in order to retrieve annotations from the input and establish annotations in the output.

Our approach extends the basic rewriting paradigm with support for the separation of concerns that involve annotations. This is motivated in the context of transformations for software re-engineering. In this area, annotations can be used to implement concerns such as layout preservation, pattern simplification, change logging, remote referencing, postprocessing directives, and access restrictions.

Keywords

Re-engineering, Annotations, Term rewriting, Separation of concerns, Aspect-oriented programming, Generic programming

1 A challenge in Cobol re-engineering

Suppose we want to renovate Cobol programs using program transformations. Typical examples are dialect conversion, GO TO elimination, data-field expansion, migration from Cobol files to a relational database, and migration from an ASCII user interface to a GUI. We assume that a rewriting-based approach is adopted for the implementation of the transformation functionality. In the case of dialect conversion, for example, we would gather rewrite rules that translate patterns of the source dialect into patterns of the target dialect. Let’s turn to the ‘side issues’: a few additional concerns regarding such renovation tooling are worth an effort.

Layout preservation The renovated sources must preserve the original comments and white spaces. Otherwise, maintenance programmers will argue that the sources were alienated. Layout preservation is also a precondition for the visualisation of code changes if we wanted to use a simple tool like diff.

Pattern simplification To ease the development of transformation functionality, the programmer should operate on a simple, core syntax for Cobol as opposed to Cobol’s complex, full syntax. To this end, Cobol programs are normalised prior to transformation. The original patterns must be restored in the output to avoid an alienation of the sources.

Change logging We want to keep track of all code changes where we ask for a precise approach as opposed to an ad-hoc approach that uses a tool like diff. Such change logging is useful to visualise affected code regions, to record changes for later reference, and to enable user intervention.

Remote referencing Given a data name in a Cobol statement, we want to access directly the corresponding declaration site. Such remote referencing is very convenient for encoding typical analyses and transformations. Otherwise, remote sites had to be repeatedly located by tree traversals.

Postprocessing directives COPY books (i.e., include files) should be expanded during preprocessing to make their content accessible within the parse trees. The COPY statements should be re-established during postprocessing on the basis of directives carried in the parse trees.

Access restrictions In the presence of expanded COPY books, it needs to be defined how to handle changes of the parse-tree regions that correspond to expanded files. Normally, there is no useful way to deal with such changes. Then, any attempt to perform such changes by rewrite rules should be refused.

It turns out that these concerns are crosscutting concerns from the point of view of rewriting. That is, a traditional implementation of such a concern will be tangled in the sense that many or all rewrite rules need to pay attention to the concern. In this paper, we provide a rewriting-based computational model that allows for the separation of concerns like those above. This model provides designated support for processing annotated parse trees. The key insight is here the following:

- Concerns like those above can be captured with annotations.
- The treatment of annotations can be largely decoupled from the ordinary, annotation-unaware rewrite rules.

Our application experience stems from using annotated parse trees in GDK [11] (i.e., C-programming with generative support for rewriting) and Laptops [13] (i.e., prological language processing). Our approach is equally suitable for fully-fledged term rewriting frameworks such as ASF+SDF [1], and it can be faithfully instantiated for object-oriented as well as functional programming.

The paper is structured as follows. In Sec. 2, we exemplify some typical shortcomings of techniques that are used for transformation technology in software re-engineering. These examples serve as a motivation of our approach. In Sec. 3, we characterise the overall computational model underlying our approach. The components of this model are explained in detail in Sec. 4–Sec. 7. Related work is discussed in Sec. 8. The paper is concluded in Sec. 9.
2 Common shortcomings

The present section illustrates a few specific techniques for the implementation of the concerns posed in the introduction. We will pinpoint typical shortcomings of transformation technology for software re-engineering, namely code tangling, persistent normalisations, low-level annotations, and restriction to tree-shaped data. It is this pool of problems which triggered our interest in a designated ‘separation of concerns’ technique. For some of the problems, cures are available in some systems (e.g., ASF+SDF, Recoder, Strafuski, Stratego), but what is lacking is a general solution to allow for the effective separation of all the relevant concerns. The subsequent sections initiate such a general approach.

Code tangling

We illustrate this shortcoming using as an example a technique that addresses the concern of layout preservation. Following a local tradition, we use the term ‘layout’ here to denote both white spaces and comments. One major option to meet the requirement for layout preservation is to incorporate all layout into the parse tree, and to maintain it during transformation so that layout can be used during unparsing. In [3, 15], this option is adopted in the following way. The original context-free grammar is expanded to include a nonterminal for layout preceding every terminal in the context-free productions. Hence, a parse tree according to the expanded grammar carries additional branches for layout all over the place. The expanded grammar is also used for rewriting, i.e., patterns for matching need to include variables to catch layout in the various positions. In this case, the concern of layout preservation is entangled in the grammar and in the rewrite rules. To give an example, we include the syntax of one form of Cobol’s MOVE statement before and after expansion in Fig. 1. (One does not need to be a Cobol expert to appreciate this example. It is enough to notice the occurrences of Layout, which were systematically added to the production.)

Persistent normalisations

Monster languages like Cobol naturally invite for normalisations to be applied prior to actual processing. If trivial productions are eliminated, then pattern-match cases are less verbose and fewer cases have to be covered anyway. In [2], a perl-based preprocessor is discussed, which performs several operations on the source code prior to parsing and rewriting. In particular, the preprocessor removes optional keywords and it replaces keywords by their ‘normal forms’. This is illustrated by a perl snippet in Fig. 2. For example, the keyword INITIAL is removed, and the variation HIGH-VALUES is normalised to HIGH-VALUE. This lexical normalisation is not reversed which implies pervasive code changes.

In fact, the more sophisticated normalisations get, the more difficult it is to reverse them. To give an example of sophisticated normalisations, we refer to ‘context-free preprocessing’ [12] where entire patterns are normalised away. A Cobol example is the following:

\[
\text{A NOT Rel-Op B } \rightarrow \text{ NOT A Rel-Op B}
\]

So the negating prefix NOT of a relational operator (e.g., “=”) is factored out to the level of Boolean expressions. An even more advanced normalisation would be about Cobol’s ‘abbreviated combined relation conditions’. This condition form allows a programmer to omit operands and operators in compound conditions while defaults of the missing elements are derived from the context. The following example illustrates how such an abbreviation can be turned into an ordinary condition:

\[
\text{IF } A = 1 \text{ OR } 2 \ldots \rightarrow \text{ IF } A = 1 \text{ OR } A = 2 \ldots
\]

In general, it is not well understood how to organise normalisations in a way that they can be reversed in the output of a transformation. This is an obstacle to using normalisations in practice. To be on the safe side, few normalisations are performed, and the code is preserved as much as possible. This implies that the benefits of normalised representations are not accessible to the programmer of transformation rules.

Low-level annotations

In the implementation of program transformations, it is quite common to use richer representation types than just the signature of the object language. There are various approaches that involve some form of annotations. These are often low-level approaches in the sense that programmers receive little support in retrieving, preserving, and establishing annotations. In particular, there are many approaches that rely on comment conventions. We will illustrate such low-level approaches using two examples, namely the use of ‘scaffolds’ to carry intermediate results through staged transformations and ‘water marks’ to keep track of obligations for postprocessing.

Let us consider a transformation engine for the folklore Y2K problem. The transformation is staged in phases including opportunities for human intervention. The Y2K engine might start from a programmer-supplied seed set for date fields. A subsequent propagation step determines the fields with the same type of usage as the data fields in the seed set. Then, code patterns are identified that involve the affected fields. Finally, the identified patterns need to be changed, and the changes need to be documented. In Fig. 3, we illustrate two of these stages. The Cobol snippet at the top shows legacy code which moves the various components of a date to data...
fields in an accept/display buffer SCREEN37. As one can see, the code assumes a fixed value 19 for the century. The Cobol snippet in the middle reflects the result of all analyses and the identification of affected code fragments. The findings of the Y2K engine are reported by a kind of comment — a scaffold according to [15]:

SCAFFOLD [ MOVE-CENT [ FILE42-YY : Identifier ]] 

That is, the Y2K engine proposes to treat the subsequent MOVE 19 ... statement as a MOVE statement that fills a CENTURY field. The year for the windowing decision is found in FILE42-YY. The Cobol snippet at the bottom shows the code that results from implementing the scaffold by a transformation. That is, using $\delta$ as the cut-off year, either 19 or 20 is moved to the century field. It is important to notice that scaffolds are like comments that can be placed anywhere in the source code. There is only limited support for locating scaffolds, for preserving them, and for identifying their scope. This is what we mean by 'low-level annotations'.

Let us also discuss the use of low-level annotations for Cobol's COPY-book expansion. COPY books (say, include files) are reusable code fragments for the DATA and the PROCEDURE DIVISION. These COPY books are normally expanded prior to transformation so that the complete source code of a program can be accessed. The expansion is reversed in the final code. COPY-book expansion is illustrated in Fig. 4. The code snippet at the top still contains a COPY statement. (As an aside, without expansion, the shown code is not proper Cobol because the period terminating the USING clause resides in the COPY book. The period at the end of the COPY statement really just terminates this statement.) The code snippet at the bottom is the result of COPY-book expansion. The content that originates from the COPY book is surrounded by comments so that postprocessing can later undo the expansion. Obviously, such ‘water marks’ are fragile. That is, a transformation might garble the comments. Also, a transformation might transform the expanded content of the COPY book, although this is normally not sensible.

Figure 5. Snippet of an algebraic specification for type-of-usage propagation for Cobol.
transformation. Also, these links will be immediately useful once we need to update data names, in particular, if this needs to be done jointly for both use and declaration sites.

3 Rewriting with annotations at a glance

We will now characterise a computational model of rewriting with annotations. This model will allow us to treat problems like the above-mentioned ones in a uniform and modular manner. The model is based on the following concepts:

- **Rewrite rules**: these are the basic computations in tree processing. A rewrite rule is a problem-specific action on a given syntactical sort. It performs tree matching and tree building.
- **Traversal schemes**: they provide means to systematically apply rewrite rules to the complete parse tree. Good candidates are innermost normalisation, or bottom-up traversal.
- **Annotations**: each node in a parse tree can be annotated. Each annotation can be organised as a vector of atomic annotations to deal with separate concerns.
- **Progression methods**: they define the annotations in the results of rewriting steps. They are applied whenever a rewrite rule does not establish an annotation by itself.
- **Access methods**: they can be used in the definition of rewrite rules in order to retrieve annotations from the input and to establish annotations in the output of a rewrite step.

The following sections detail these concepts.

4 Rewrite rules and rewrite steps

In our model, rewrite rules correspond to the most basic pieces of functionality, i.e., to the building blocks of program transformations. Typically, rewrite rules perform tree matching and building. In addition, they might invoke subcomputations on parts of the tree, and they might be constrained by side conditions. Below, we will see that rewrite rules can also perform actions to access annotations. We use the term *rewrite step* to denote an application of a rewrite rule in the course of a complete traversal over a parse tree.

The notion of a rewrite rule can be incarnated as follows:

**Term rewriting** just a rewrite rule.

**OOP** a function using a term API.

**Functional programming** a function on term types.

**C** a function using a term API.

To illustrate the link between rewrite rules and rewrite steps, let us consider a simple example of a rewrite rule written down in the notation of term rewriting:

\[ a(x_1, c(x_2, x_3)) \rightarrow a(x_1, f(x_3, x_2)) \]

Here, \(a\), \(c\), and \(f\) are function symbols whereas \(x_1\), \(x_2\), and \(x_3\) are tree variables. In Fig. 6, we visualise the application of this rewrite rule to an input tree in a rewrite step. The arrows in the figure illustrate how function symbols and subtrees of the input reappear in the output. These sharing relations between input and output will be useful to guide the propagation of annotations. The sharing relations for subtrees are immediately implied by the occurrence of variables on both sides of a rewrite rule. The sharing relations for function symbols are not unambiguously specified by a rewrite rule itself. (Think of a function symbol that occurs once on the left-hand side, and twice on the right-hand side. Or think of a function symbol on the right-hand side that is meant to be ‘fresh’, although the same function symbol occurs accidentally on the left-hand side.) So we advocate a notation for rewrite rules which makes the sharing relations for function symbols explicit. In the following revision of the above rewrite rule, we tag the two occurrences of \(a\) by \(t_a\) to express that we mean the same \(a\) on both sides of the rewrite rule:

\[ a(t_a(x_1, c(x_2, x_3))) \rightarrow a(t_a(x_1, f(x_3, x_2))) \]

It remains to define how ordinary rewrite rules are applied to annotated parse trees. We also need to clarify how rewrite steps are interleaved with actions for the propagation of annotations. Furthermore, we need to extend the notion of a rewrite rule to be able to access annotations. The following sections cover all these topics.

5 Annotated parse trees

We assume a very simple scheme for annotating parse trees. That is, we pair each ordinary node with an extra annotation node. This is illustrated in Fig. 7 for the input of the rewrite step from Fig. 6. An annotation-unaware rewrite rule can be matched against an annotated parse tree without ado or penalty: we skip the extra pairing level and we navigate to the right component of the “( )” node before function symbols are compared.
It will be vital that each annotation is potentially a vector of atomic annotations. This will allow us to model different annotation concerns, e.g., layout preservation, and pattern simplification, and change logging, and so forth. Common types of atomic annotations are the following:

- Boolean for flagging subtrees.
- Natural numbers for counters.
- Term types to backup subtrees.
- Pointer types to link nodes in the parse tree.

(We keep the term ‘parse trees’ despite the potential of pointers.)

Let us briefly indicate alternative formats for annotated parse trees. Firstly, one could argue in favour of having specific types of annotations for each syntactical sort. This would add opportunities for optimisation and static checks. A corresponding elaboration is a perfect subject for future work. Secondly, one could attempt to use extra branches for normal nodes in a parse tree as opposed to our extra nodes which pair ordinary nodes and annotations. This format is indeed used in existing approaches for layout preservation [3, 15, 4, 17]. The mere placement of these added branches assigns a fixed meaning to the annotations, namely ‘the layout preceding a token’. Such a fixed meaning is not acceptable for our purposes because we want to model various concerns with annotations, and not just a specific form of layout preservation.

As for the concerns from the introduction, the following types of annotations are appropriate:

**Layout preservation** We use chunks of layout, say strings. There are different ways how layout can be grouped in annotations. One approach is to only annotate leaf nodes, i.e., tokens, and to capture the layout preceding a token as its annotation.

**Pattern simplification** We use the annotation to backup the original subtree. The normalised tree and the original tree might share subtrees. This can be expressed if we assume that annotated parse trees are actually directed acyclic graphs.

**Change logging** We maintain historical trees for each node. This is illustrated in Fig. 8. The change log might refer to ordinary subtrees to express sharing of subtrees in historical trees and the current tree. Such sharing promotes both an efficient representation and a precise specification of changes.

**Remote referencing** The annotation is a link to a declaration site. Here, it becomes indispensable that we allow for directed acyclic graphs (or even more general graphs) for the representation of annotated parse trees.

**Postprocessing directives** The annotation is basically the COPY statement. There are however different ways to place the annotation in the parse tree. If the COPY book hosts a syntactical unit, we can place the annotation at the node that roots the content from the COPY book. Otherwise, we have to flag all tokens that originated from the COPY book.

**Access restrictions** Recall that this concern checks that changed subtrees do not reside in COPY books. Hence, this concern does not necessitate a further annotation since it can be defined in terms of the aforementioned annotations for reversing COPY-book expansion.

The subsequent two sections will clarify how to handle annotations.

### 6 Progression methods

Even if a rewrite rule, by itself, is not concerned with annotations, still the annotations in the result of a corresponding rewrite step need to be defined. Here, a crucial insight enters the scene. In our approach, not just the annotation types but also the rules for their progression can be defined by the programmer. It is important to notice that progression can be defined differently for each atomic annotation of an annotation vector.

There are three overall schemes for progression:

**Initialisation** An annotation in the output is defined without any reference to an annotation in the input.

**Propagation** An existing annotation is propagated from the input to the output of a rewrite step on the basis of sharing relations. This propagation might involve side conditions, and an update of the propagated annotation.
Synthesis An annotation is computed while observing both the input and the output of a rewrite step, including the involved tags and annotations. This is the most general scheme.

Some detailed options for the scheme of propagation are illustrated in Fig. 9. The copy option is suitable if a given function symbol in the output also occurred in the input, and the annotation $v_1$ should be preserved as is. A good example is a flag the state of which should be maintained. The update option is in place if the input annotation $v_1$ should not be preserved as is in the output, but it should be adapted to $v_1'$. A good example is incrementing a counter annotation whenever a subtree is affected by a rewrite step. The reset option is in place if a certain default is favoured for the annotation in the output, e.g., 0. The generate option is used when an annotation serves as a kind of unique node identifier. Finally, the guard option models that a progression method can possibly fail in order to report on an inconsistent rewrite step. The options in the figure can also be combined in various ways. For example, we might deal with a partial update, that is, an update which also involves a guard.

Progression methods for initialisation and propagation can be defined very easily. We assume that these methods only need to cover constructed annotation nodes in the output tree, that is, nodes which are associated with function symbols on the right-hand side of the rewrite rule. Shared subtrees are reused as is. Then, the definition of progress for a given function symbol comes down to providing two actions for two different cases:

1. The right-hand side function symbol is not shared. That is, there is no corresponding function symbol on the left-hand side of the rewrite rule. We need to perform an action for initialisation.

2. The right-hand side function symbol is shared. That is, there is a candidate annotation from the left-hand side. We need to perform an action for propagation.

In Fig. 10, we define these two actions for a few progression methods in Prolog. The shown predicates are part of a Prolog implementation of our approach for processing annotated parse trees. For a homogeneous situation, we always define two predicates, even if some of the Prolog clauses immediately resort to fail. The defined methods basically exercise all the options from Fig. 9.

The more general progression scheme for synthesis is of use for more sophisticated concerns. Its generality allows us to compute some closure on the input or the output of a rewrite step. For example, we can identify all atomic annotations of a certain type in the input which do not reappear in the output as drafted by the rewrite rule. Such an analysis is useful for a form of layout preservation where vanishing comments are relocated to the output tree.

To conclude this section, the concerns from the introduction of the paper can be modelled using the following progression methods:

**Layout preservation** In the most simple case the ‘reset’ option is in place. This will achieve the amount of preservation in previous work on the subject. We can also slightly improve the amount of preservation by using the ‘copy’ option. This can be even further improved by a designated progression method as indicated above.

**Pattern simplification** We adhere to the ‘update’ option where we invalidate the original tree, that is, the normalisation becomes persistent as a result of changing the tree.

**Change logging** We adhere to the ‘update’ option where we append the input tree to the current change log for this node.

**Remote referencing** Basically, the ‘copy’ option is appropriate.

**Access restrictions** We need to check that the output tree as proposed by the application of a rewrite rules does not attempt to garble the content of a COPY book. We use a designated progression method for this purpose. It tests if pattern-matched fragments of COPY book content reappear in the output tree.

---

**Figure 9.** Different options for the propagation of a given atomic annotation $v_1$ in the course of the rewrite step (cf. Fig. 6).

**Figure 10.** Prolog encoding of typical ingredients for progression.

```prolog
% Propagation by copying
copy1(_) :- fail.
copy2(Anno,Anno).

% Integer increment
incl1(_) :- fail.
icl2(IO,II) :- II is IO + 1.

% Initialise or reset to 0
zerol(0).
zero2(_,0).

gen1(Sym) :- gensym('_',Sym).
gen2(Anno,Anno).

% Generate fresh annotation if needed
noreuse1(false).
noreuse2(true,_) :- !, fail.
noreuse2(Anno,Anno).
```

---

1. The right-hand side function symbol is not shared. That is, there is no corresponding function symbol on the left-hand side of the rewrite rule. We need to perform an action for *initialisation*.
7 Access methods

For several typical annotation concerns, rewrite rules are predominantly unaware of annotations. For example, layout should be preserved largely for free. Also, access restrictions should definitely be enforced without the programmer’s intervention. There are however concerns which require certain rewrite rules to be aware of annotations. A good example is fine tuning of comment preservation. Then, comments need to be moved from the input to the output, which can involve very specific arguments related to the semantics of the language and the transformation rules at hand.

Annotation-aware rewrite rules can access the annotations of the involved nodes on the basis of tree variables and tags of function symbols. To this end, we assume access methods for retrieving and establishing annotations. These access methods reside in our ‘annotation library’ just as the progression methods.

In the following, we illustrate annotation-aware rewrite rules via an elaboration of our running example:

\[
ax \triangleright (x_1, ct(x_2, x_3)) \rightarrow ax \triangleright (x_1, f(x_2, x_3))
\]

retrieve comments = tc.getComments()
establish tf.putComments(comments)

In this example, we retrieve the comments from the vanishing function symbol \(c\) and we establish the comment annotation for the introduced function symbol \(f\). As we can see, annotations that are retrieved from the input can be used in the method invocations that establish annotations in the output. In principle, retrieved annotations can also be used in constructing the output tree itself.

To conclude our discussion of handling annotations, we emphasise that both progression and access methods are by no means restricted to the nodes that correspond to tags in a rewrite rule. In fact, tagged symbols and shared subtrees serve merely as anchors for walking over the annotated parse trees. For example, the access methods in the above example do not necessarily retrieve comments from \(tc\) or establish comments at \(tf\). The relevant annotations are actually placed at tokens below \(tc\) and \(tf\) depending on details of the layout concern. In general, both progression and access methods can indeed traverse into subtrees including access to annotations in these subtrees. Depending on the incarnation of our approach, a method can also walk upwards the tree so that we can basically reach all nodes in the parse tree regardless of our current window.

8 Related work

Attribute grammars vs. rewriting

Our approach is related to attribute grammars [10] (AGs). While our computational model is predominantly rewriting-based, we indeed borrow certain elements of both the basic AG formalism and some of its vital extensions (e.g., [7, 8]). Understanding the precise correspondence is a useful exercise.

AGs assign a meaning to context-free grammars by attributing parse trees. The values of the attributes are computed by means of semantic rules associated to the context-free productions. AGs are typically applied for semantic analysis and intermediate code generation. AGs are, in our experience, much less suited for transformations than rewriting. This can be substantiated as follows:

Asymmetry between input and output The input is defined in terms of the context-free grammar while the output is defined by some root attribute. If this distinguished attribute happens to be of the type of a syntactical domain, then one can view an AG as describing a transformation. In rewriting, a symmetric situation is reflected by the fact that both left- and right-hand side of a rewrite rule are terms. Hence, a rewrite rule encodes a (piece of a) transformation.

Lack of tree matching / building This rewriting expressiveness is very convenient. In the AG setting, transformation rules cannot be encoded directly, but they rather need to be scattered over the productions of the underlying context-free grammar.

Lack of normalisation / traversal Attribute evaluation is driven by attribute dependencies for the nodes in a parse tree. This is useful if the different kinds of nodes carry attributes of different types, and there are rich, non-local dependencies between the attributes. This is normally not the case for transformations. They are more appropriately performed by exhaustive application of some rewrite rules all over the tree.

We should note that our annotations are not quite like attributes in AGs. This is because attributes are immutable place holders in the mathematical sense, while we assume that annotations progress for each rewrite step. Otherwise, our concepts are inspired by ideas in the AG field. That is, our progression methods are similar to symbolic computations and forms of remote access in [8]. Our consideration of graphs instead of trees (recall links as annotations) is similar to the amalgamation of the purely declarative AG formalism with references as in [7].

Sophisticated rewriting approaches

There are extensions of term rewriting which address some of our concerns. Most notably, there is work on rewriting with layout [3, 15, 4, 17]. In the most simple case, no extension of the rewriting machinery is assumed [3, 15], which implies some code tangling as discussed in Sec. 2. A more integrated approach is the one in [17] where overlays for term matching and building are used. While the primary term constructors contain layout positions, these overlays do not. Thereby, readability of rewrite rules is retained.

An approach towards built-in support for layout preservation is presented in [4]. Here, the rewrite engine is aware of layout positions in a way that rewrite rules again can omit layout positions from patterns.

Our work shows that layout preservation is just an instance of a more general problem: the annotation of the object programs in a transformation with data needed for additional concerns. The important contribution of our approach is that separate concerns can be effectively defined because annotation types, progression methods, and access methods are all programmer-definable. To give an example, while previous work on layout preservation assumes empty layout positions in building a pattern in a rewrite rule, this is just one option of a progression method. There is another extension of rewriting which is related to our work, that is, the work on origin tracking [6]. Origins are relations between subterms of intermediate terms which occur during rewriting. A concern like our change logging could be handled in some way with origin tracking. However, origin tracking is limited in so far that it does not involve programmer-definable elements, in particular, no extra annotations. The programmer can only observe the origins for the intermediate terms which occur during rewriting.
**Pattern matching on abstract data types**

In a way, we are concerned with the problem that different functionality favours different views on the transformed trees. In most rewrite rules, we would like to abstract from annotations, while the definition of progression methods is aware of the complete representation types, that is, trees including annotations. In functional programming, a similar problem has been studied: pattern matching for abstract data types [16, 5, 14] (aka views). Views allow one to use the functions of an ADT in pattern matching as if these corresponded to proper constructors. Our approach involves an essential element that is not present in this work: tree annotations are potentially propagated from the input to the output of a rewrite step as defined by progression methods.

**9 Concluding remarks**

Annotations are needed in practical applications of rewriting, in particular in the context of the implementation of transformations for software re-engineering. Rewriting with annotations is easy to integrate into a standard rewriting setting. Our approach can be seen as a scheme to open up rewrite engines for performing actions on annotated trees in parallel with ordinary tree matching, building, and normalising. This is in contrast to a situation where a rewrite engine would anticipate specific concerns. Our approach effectively disentangles ordinary rewriting and annotation concerns. In particular, rewrite rules do not need to catch annotations when matching trees, and they also do not need to supply annotations when building trees. With access methods, rewrite rules can retrieve and establish annotations if this is necessary. With progression methods, basic rewrite steps are complemented in a generic way so that annotations are also propagated or initialised.

In our ongoing work, we aim at a seamless support for the approach in the language processing environments GDK, Laptoib, and Strafunski. To this end, a few conceptual issues deserve further research. Firstly, the typing of progression methods is somewhat involved, and needs to be formalised. Secondly, the actual design space for progression methods is still largely unexplored. Thirdly, the composition of separate annotation concerns has to be formalised including the possible interaction of the concerns.

**Acknowledgement**

We are grateful for discussions with Paul Klint, Wolfgang Lohmann, Günter Riedewald, and Jurgen Vinju. We are particularly indebted to the four anonymous referees of the GI AOSD 2003 workshop for their very encouraging feedback.

**10 References**


