Parse-Tree Annotations Meet Re-Engineering Concerns

Jan Kort [1] and Ralf Lämmel [2]

[1] Universiteit van Amsterdam, Kruislaan 403, NL-1098 SJ Amsterdam
[2] Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Email: kort@science.uva.nl, ralf@cwi.nl

Abstract

We characterise a computational model for processing annotated parse trees. The model is basically rewriting-based with specific provisions for dealing with annotations along the ordinary rewrite steps. Most notably, there are progression methods, which define a default for annotating the results of rewriting. There are also access methods, which can be used in the rewrite rules in order to retrieve annotations from the input and to 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 where annotations can be used to implement concerns such as layout preservation and reversible preprocessing.

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. To implement the corresponding transformation functionality, rewriting-based systems such as ASF+SDF [2] or TXL [6] are quite suitable. 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. Here they are:

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 would also be a precondition for the visualisation of code changes if we wanted to use a simple tool like `diff`.)

Pattern simplification. To hide some of Cobol's complexity, the programmer should operate on a simple core syntax for Cobol as opposed to Cobol's full syntax. To this end, Cobol programs are normalised prior to transformation. The original patterns must be restored in the output to avoid a drastic alienation of the sources.

Change logging. We want to record all code changes in a precise, syntax-aware manner. Such change logging is useful for visualising affected code regions, recording changes for later reference and enabling 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 convenient for encoding typical analyses and transformations. Otherwise, remote sites had to be repeatedly located by tree traversals.

Post-processing 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 post-processing 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 arose from expansion. If there is no useful way to deal with such changes, then any attempt of rewrite rules to perform such changes should be refused.

Plan of attack — separation of concerns It turns out that these are *crosscutting* concerns in a rewriting context. (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 of our approach is the following:

- The data for concerns like those above can be captured as annotations of the parse trees that are rewritten.
- The treatment of annotations can be largely decoupled from the ordinary, annotation-unaware rewrite rules.

Standard production format; without mentioning layout

```
move-statement = "MOVE" ( "CORRESPONDING" | "CORR" ) identifier
                   "TO" {identifier}+
```

Pervasive use of layout positions in the productions

```
move-statement = Layout "MOVE" ( Layout "CORRESPONDING" | Layout "CORR" ) identifier
                 Layout   "TO" {identifier}+
```

Figure 1. From an ordinary Cobol grammar to a layout-aware one.

We have experimented with annotated parse trees in *GDK* [14] (i.e., *C*-programming with generative support for rewriting) and *Laptob* [16] (i.e., prological language processing). The ultimate goal is to integrate this approach with a strongly typed, powerful rewriting framework such as *ASF+SDF* [2], and to also instantiate it seamlessly for object-oriented as well as functional programming.

The paper is structured as follows. In Sec. 2, we exercise typical techniques that occur in transformation technology for software re-engineering. Weaknesses of the techniques serve as a motivation for our approach. In Sec. 3, we characterise the overall computational model underlying our approach, while details follow in Sec. 4–Sec. 7. Related work is discussed in Sec. 8, and the paper is concluded in Sec. 9.

2. Common shortcomings

We will now review a few specific techniques for the implementation of the concerns posed in the introduction. We will argue that these techniques (and hence corresponding transformation technology for software re-engineering) exhibit shortcomings such as code tangling, persistence of normalisations, low level of abstraction, and restrictiveness regarding parse-tree formats. It is this *pool of problems* which triggered a designated 'separation of concerns' technique. Most modern transformation technologies such as ASF+SDF, Recoder, Strafunski, Stratego, TXL attack some of these problems, but there is no general approach to allow for the effective separation of all the relevant concerns. (This paper works towards such a general approach.)

Code tangling We illustrate this shortcoming with 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 [21], 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, parse trees comprise 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 simplify the implementation of transformations. If some variations are eliminated or simplified, fewer or simpler cases have to be covered by the transformation rules. In [3], a simple `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. The more sophisticated normalisations get, the more difficult it is to reverse them. The following rewrite step illustrates context-free (as opposed to lexical) normalisation for Cobol [15]:

$$A\ \texttt{NOT}\ \mathit{Rel\text{-}Op}\ B \quad \rightsquigarrow \quad \texttt{NOT}\ A\ \mathit{Rel\text{-}Op}\ B$$

So the negating prefix `NOT` of a relational operator is factored out to the level of Boolean expressions. A more advanced normalisation would be about Cobol's abbreviated

```
...
$line =~ s/(\s)INITIAL(\s|\.)/$1$2/g;
$line =~ s/(\s)GO\s+TO(\s|\.)/$1GO$2/g;
$line =~ s/(\s)INDEXED\s+BY(\s|\.)/$1INDEXED$2/g;
$line =~ s/(\s)SPACES(\s|\.)/$1SPACE$2/g;
$line =~ s/(\s)ZEROS(\s|\.)/$1ZERO$2/g;
$line =~ s/(\s)ZEROES(\s|\.)/$1ZERO$2/g;
$line =~ s/(\s)VALUES(\s|\.)/$1VALUE$2/g;
$line =~ s/(\s)LOW-VALUES(\s|\.)/$1LOW-VALUE$2/g;
$line =~ s/(\s)HIGH-VALUES(\s|\.)/$1HIGH-VALUE$2/g;
...
```

Figure 2. Perl snippet for lexical normalisations.

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 expanded accordingly:

$$\texttt{IF}\ A\ =\ 1\ \texttt{OR}\ 2\ \ldots\ \rightsquigarrow\ \texttt{IF}\ A\ =\ 1\ \texttt{OR}\ A\ =\ 2\ \ldots$$

In general, it is not well understood how to implement normalisations in a way that they can be reversed later. 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 working re-engineer.

Low-level annotations In the implementation of program transformations, it is quite common to use some form of annotations to keep track of all sorts of intermediate information. The common annotation mechanisms tend to require from the programmer to operate at a low level of abstraction, e.g., by relying on comment conventions or an unrestricted attribute scheme. We will discuss two examples of using low-level annotations, one to carry intermediate results through staged transformations and another to keep track of obligations for post-processing.

The first example deals with a staged transformation for the Y2K benchmark. (Here, it is not necessary to be an Y2K expert. The example is only meant to illustrate how different kinds of comments direct and reflect on a staged transformation.) The following stages are relevant. The Y2K engine starts 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 that moves the various components of a date to data fields in an accept and display buffer `SCREEN37`. (As one can see, the legacy 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. This is called a scaffold according to [21], or a maintenance hot spot according to [7]:

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

The transformation stage reads this comment as a proposal to treat the `MOVE 19 ...` statement following the scaffold as a `MOVE` statement that fills a `CENT`ury 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

Original code with Y2K problem

```
  ...
  MOVE FILE42-DD TO SCREEN37-DD.
  MOVE FILE42-MM TO SCREEN37-MM.
  MOVE FILE42-YY TO SCREEN37-YY.
  MOVE 19        TO SCREEN37-CENT.
  ...
```

Code with scaffold for scheduled windowing adaptation

```
  ...
  MOVE FILE42-DD TO SCREEN37-DD.
  MOVE FILE42-MM TO SCREEN37-MM.
  MOVE FILE42-YY TO SCREEN37-YY.
  SCAFFOLD [ MOVE-CENT [ FILE42-YY : Identifier ]]
  MOVE 19        TO SCREEN37-CENT.
  ...
```

Adapted code with maintenance comment

```
  ...
  MOVE FILE42-DD TO SCREEN37-DD.
  MOVE FILE42-MM TO SCREEN37-MM.
  MOVE FILE42-YY TO SCREEN37-YY.
* BEGIN MY-Y2K-ENGINE: MOVE-CENT FILE42-YY
  IF FILE42-YY > 68
   MOVE 19       TO SCREEN37-CENT
  ELSE
   MOVE 20       TO SCREEN37-CENT.
* END MY-Y2K-ENGINE: MOVE-CENT FILE42-YY
  ...
```

Figure 3. Two steps in a staged windowing transformation for the Y2K problem. We use scaffold comments in the intermediate result.

is, using `68` 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 are no higher-level idioms for locating scaffolds, for preserving them, and for identifying their scope.

As another example of low-level annotations, we consider the link between pre- and post-processing with focus on Cobol's `COPY` books. (These are like include files that comprise reusable code fragments for Cobol programs.) `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 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 shows the result of expansion. The content that originates from the `COPY` book is surrounded by special comments so that post-processing can later undo the expansion. While this approach is effective in practice, it is obviously entirely unsafe. That is, a transformation might garble the comments. Also, a transformation might end up transforming the inlined content of the `COPY` book, although this is normally not sensible.

Before COPY book expansion

```
...
PROCEDURE DIVISION USING SCREEN01
                         SCREEN02
                         COPY LINK-REST.
...
```

After COPY book expansion

```
...
PROCEDURE DIVISION USING SCREEN01
                         SCREEN02
* BEGIN COPY-BOOK EXPANSION LINK-REST
                         SYS-PARA
                         USR-PARA
                         OPT-CODE
                         RET-CODE.
* END COPY-BOOK EXPANSION LINK-REST
...
```

Figure 4. A Cobol program that uses a COPY book for a reusable list of subprogram arguments. The expanded code is scoped by comments.

Restriction to tree-shaped data Especially in the context of rewriting, tree shape is taken as an axiom. That is, a parse tree is really nothing more than a tree as opposed to a more general graph data structure. So rewrite rules can observe subtrees of a given tree, but there is no way to navigate from a given node A to a 'related' node B if A does not root B. The technique of a symbol table is typically used to compensate for this restriction. However, the presence of name spaces or the need to modify remote nodes require further provisions. The restriction to tree-shaped data is an obstacle to simple encodings of some recurring scenarios in software re-engineering, e.g., the direct navigation from use to declaration sites.

Let us illustrate the restriction to tree shape by using a fragment of the specification for a Y2K-like project discussed in [11]. The specification is executable in the ASF+SDF Meta-Environment [2]. In Fig. 5, we show two conditional rewrite rules that specify type-of-usage propagation. Given a set of affected data names *Sofar*, the shown rules identify more affected fields and accumulate them as a parameter *Found*. Identification of the same type of usage is based on looking at Cobol patterns that use two data names; here MOVE statements. We include a data name, if it is not yet in *Sofar* while its companion data name in the MOVE statement is in *Sofar*. For the comparisons of data names to be correct, we have to assume that all the data names are qualified. (Think of Cobol's nested group fields.) This would normally require a pervasive transformation. We might also perform name qualification on the fly by an expensive traversal of the program's DATA DIVISION.

$$[1]\quad \frac{Id_1 \in Sofar = true,\ Id_2 \in Sofar = false}{Propagate(\textbf{MOVE}\ Id_1\ \textbf{TO}\ Id_2, Sofar, Found) = Found \cup \{Id_2\}}$$

$$[2]\quad \frac{Id_1 \in Sofar = false,\ Id_2 \in Sofar = true}{Propagate(\textbf{MOVE}\ Id_1\ \textbf{TO}\ Id_2, Sofar, Found) = Found \cup \{Id_1\}}$$

Figure 5. Snippet of an algebraic specification for type-of-usage propagation for Cobol.

If we do not restrict ourselves to tree shape, then we can perform name resolution once and for all before all other phases, and we can keep track of the resolved names using links between use and declaration sites for later reference. This will not involve any pervasive transformation. Also, these links will be immediately useful once we need to update data declarations and data references.

3. Rewriting with annotations at a glance

We will now characterise a rewriting-based computational model for processing parse trees or abstract syntax trees with provisions for annotations. This model allows us to treat problems like the above-mentioned ones in a uniform and modular manner. The model is based on the following ingredients:

- *Rewrite rules*: these are the basic computations in tree processing. A rewrite rule is a programmer-defined problem-specific action on a given syntactical sort. It performs tree matching and tree building.
- *Traversal schemes*: they are means to systematically apply rewrite rules to the complete parse tree. A good candidate is innermost normalisation as widely used in rewrite engines. We might also consider user-definable traversal schemes as in strategic programming [17].
- *Annotations*: each node in a parse tree can be annotated. We view annotations as property lists (say, name/value pairs) so that we can deal with separate concerns.
- *Access methods*: the programmer uses these methods 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.
- *Progression methods*: the programmer defines or selects these methods in order to set up the default scheme for annotating the results of rewriting. This default scheme can be overridden by using access methods in rewrite rules.

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. Operationally, 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. We will later 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. We will later see that the ordinary rewrite steps must be intertwined with actions for progressing with annotations.

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

Term rewriting	just a rewrite rule.
OOP	a visitor object.
Functional progr.	a function on algebraic datatypes.
C	a function on a designated term API.

Here is a simple example of a rewrite rule written down in the notation of term rewriting:

$$a(X_1, c(X_2, X_3)) \to a(X_1, f(X_3, X_2))$$

Here, a, c, and f are function symbols whereas X_1, X_2, and X_3 are tree variables. In Fig. 6, we visualise a corresponding rewrite step, i.e., the application of this rewrite rule to a specific input tree. The arrows in the figure illustrate how function symbols and subtrees are shared by input and output. This *sharing* relation between input and output will be useful to guide the automatic propagation of annotations.

The sharing relation for entire subtrees is immediately implied by the occurrence of variables on both sides of a rewrite rule. The intended sharing relation for nodes (say, function symbols) relies on tagging. Implicitly, a function symbol is tagged by itself. So a function symbol occurring both on the left- and the right-hand side is shared by default (unless the same symbols occurs several times on a given side). Explicit tags can be used to disambiguate multiple occurrences, to disable implicit sharing, and maybe even to express sharing for different function symbols. In the following revision of the earlier rewrite rule, we tag the two occurrences of a differently. Hence, a is not shared anymore between left- and right-hand side:

$$a{:}t_a(X_1, c(X_2, X_3)) \to a{:}t_{a'}(X_1, f(X_3, X_2))$$

It remains to define how ordinary rewrite rules are applied to annotated parse trees. We also need to clarify how rewrite steps are completed by actions for progressing with 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.

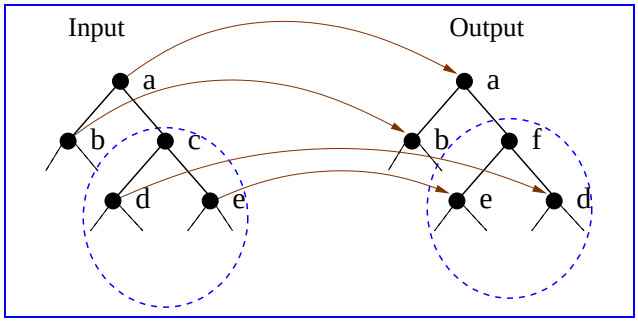

Figure 6. A rewrite step where we show the sharing relation (cf. arrows), and we focus on the changed subtree (cf. circles).

5. Annotated parse trees

We assume a very simple data structure for annotated parse trees (or annotated abstract syntax trees, and others). That is, each ordinary node in the parse tree can be annotated by a property list. This is illustrated in Fig. 7 for the input of the rewrite step from Fig. 6. The rewrite engine can now match a rewrite rule, which does not care about annotations, against an annotated parse tree very easily: the extra pairing level is skipped and only the right component of the "( )" node is considered for comparing function symbols. A useful refinement of the given format is to restrict annotations to specific, possibly optional properties depending on the node type. This adds opportunities for static checks and optimisation.

Separation of concerns It is indeed vital that nodes are annotated by property lists (say, name/value pairs) as op-

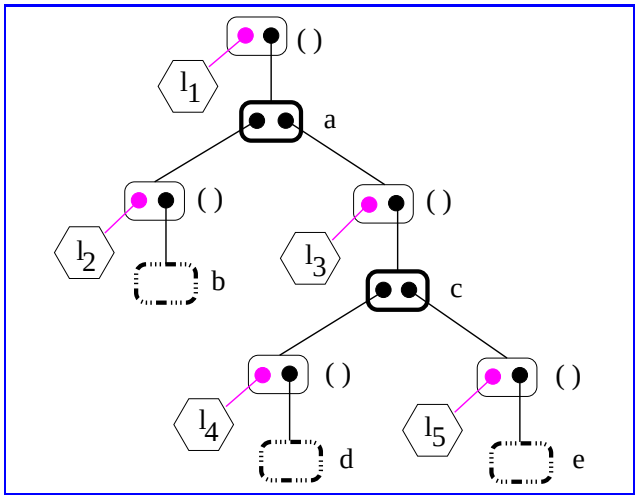

Figure 7. An annotated parse tree. There are extra nodes ("( )") for pairing ordinary nodes and annotations, say property lists.

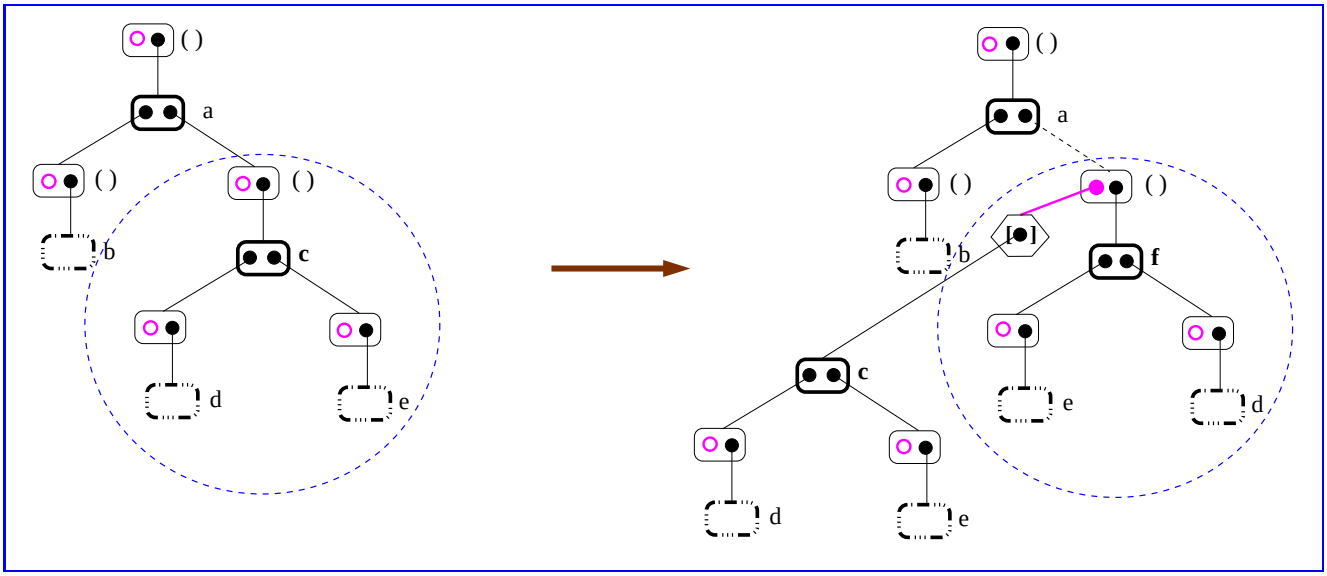

Figure 8. Change logging for Fig. 6 using lists of parse subtrees as annotation to keep track of history. The non-filled bullets indicate absence of annotations. Notice the added annotation in the result. It corresponds to the focused subtree in the input that is changed. For simplicity, potential sharing is not visualised.

posed to one blob of annotation. This allows us to effectively separate different annotation concerns, e.g., layout preservation, *and* pattern simplification, *and* change logging, *and* so forth. Common types of annotations are the following:

- Boolean for status markers.
- Natural numbers for counters.
- Parse-tree types to back up subtrees.
- Pointer types to link nodes in the parse tree.

(We keep the established term 'parse *trees*' despite the potential of pointers.)

Sample 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 turned into annotations. A simple and effective approach is to only annotate leaf nodes, i.e., tokens, and to turn the layout preceding a token into its annotation.
- *Pattern simplification*. We use the annotation to back up the original subtree. We might want to assume that parse trees are actually directed acyclic graphs. This allows us to keep track of the sharing relation between original tree and simplified tree.
- *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 between historical trees and the current tree. Such sharing is more efficient in storage, and it also specifies the changes more precisely.
- *Remote referencing*. The annotation is a link to a declaration site. Here, it becomes indispensable that we allow for at least directed acyclic graphs for the representation of annotated parse trees.
- *Post-processing directives*. The annotation is basically the `COPY` statement. If the `COPY` statement comprises a syntactically valid fragment (without prior expansion), then we can place the annotation at the node that roots the content from the `COPY` book. Otherwise, we have to flag (say, annotate) all tokens that originate from the `COPY` book.
- *Access restrictions*. Recall that this concern is meant to refuse any attempt to modify the inlined content of `COPY`-books. According to the previous concern, the mere information about expanded tokens and subtrees is already available. Hence, this concern does not necessitate more annotations.

This finishes our discussion of the data structure for annotated parse trees. It remains to explain how annotated parse trees are processed, more specifically, how the annotations are retrieved, established, preserved, etc. The subsequent two sections serve this purpose.

6. Progression methods

Even if a rewrite rule, by itself, is not concerned with annotations, we have to define how annotations from the input possibly carry over to the output, and how missing annotations are constructed if possible. 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.

Progression can be defined differently for each property that might possibly occur in the property list. For each annotation concern, there are the following overall schemes of progression:

- *Initialisation*. If a node in the output needs to be annotated, then we simply opt for a constant.
- *Propagation*. If nodes are shared between left- and right-hand side, and the left-hand side node carries a suitable annotation, then the output annotation can be computed from the input annotation, maybe constrained by side conditions.
- *Synthesis*. An output annotation is computed while observing the entire input and output of a rewrite step. This includes subtrees, involved tags and annotations. This is the most general scheme.

Some options for the scheme of propagation are illustrated in Fig. 9. We focus on a node ("a") shared between input and output tree. We look at a single property p_1. The *copy* option is used when the input annotation p_1 should be preserved as is, e.g., line-number information. The *update* option is suitable if the input annotation p_1 should be transformed (by a function f), e.g., incrementing a counter whenever a subtree is rewritten. The *reset* option is suitable if a certain default is favoured for the output annotation (e.g., 0) regardless of any available input annotation. The *generate* option is used when a kind of unique node identifier is needed. Finally, the *guard* option is used when a predicate is meant to test for consistency of rewrite step as far as annotations are concerned.

Progression methods for initialisation and propagation can be defined and implemented very easily. To this end, we make two modest assumptions. Firstly, we only consider newly constructed output nodes while shared subtrees are preserved including their annotations. Secondly, properties are not optional, that is, a certain type of node is either supposed to be annotated by a given property or not. (We can still use an 'empty' annotation to denote the absence of a proper annotation.) Then, the definition of progress for a given output node (say, function symbol) comes down to providing two actions for two different cases:

1. The function symbol is not shared between left- and right-hand side of the rewrite rule. We need to perform an action for *initialisation*.

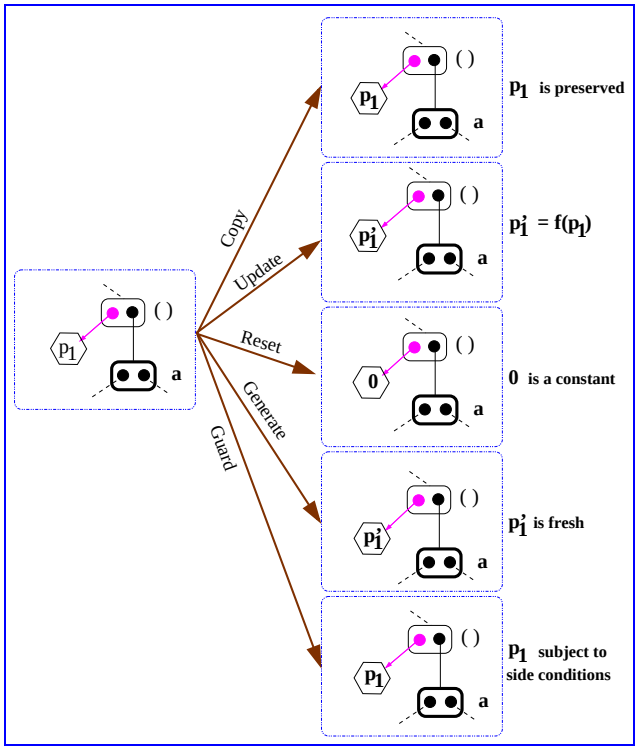

Figure 9. Different options for the propagation of an annotation p_1 of the root node from Fig. 6.

2. The 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. (Recall that we use Prolog and *Laptob* [16] to experiment with annotated parse trees.) The first predicate defines initialisation (e.g., `copy1`) and the second defines propagation (e.g., `copy2`). The listed methods basically exercise all the options from Fig. 9, and they add a fall-back solution for the non-sharing case. There are several ways in which some of the methods might deliberately fail in order to avoid inconsistent annotations.

The progression scheme for *synthesis* is really strictly more general. Its generality allows us to affect annotations of other than just newly constructed nodes, and to observe annotations of other than just nodes shared between input and output trees. For example, we can identify all input annotations of a certain type which do not reappear in the output tree as constructed by a rewrite rule. This analysis is immediately useful for a form of layout preservation where otherwise vanishing comments from the input tree are relocated to the output tree.

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

% Integer increment
inc1(_)     :- fail.
inc2(I0,I1) :- I1 is I0 + 1.

% Initialise or reset to 0
zero1(0).
zero2(_,0).

% Generate fresh annotation
gen1(Sym)   :- gensym('_',Sym).
gen2(_,Sym) :- gensym('_',Sym).

% Block progression for annotation true
noreuse1(false).
noreuse2(Anno,Anno) :- not(Anno==true).
```

Figure 10. Prolog encoding of some progression methods. They are invoked by the rewrite engine to complete ordinary rewrite steps.

Sample concerns from the introduction The following progression methods are appropriate:

- *Layout preservation*. We adhere to the 'reset' option, i.e., no reuse of layout for newly constructed nodes. This corresponds to the amount of preservation in previous work on layout preservation. Improvements can be achieved using the 'copy' option or a designated method based on the general scheme of synthesis — as indicated above.
- *Pattern simplification*. We adhere to the 'reset' option as there is no obvious way to make use of the back-up after the corresponding simplified tree was changed.
- *Change logging*. We adhere to the 'update' option, i.e., we append the input tree to the change log for the node.
- *Remote referencing*. When a tree with outgoing remote references is rewritten, then the remote references have to be redirected. The programmer of the rewrite rules can be made responsible for redirection, but it is safer to leave direction to a designated progression method.
- *Access restrictions*. A designated progression method checks that the output tree does not attempt to garble the inlined content of a `COPY` book. So the method serves as a guard. (There are other solutions to this problem, which do not require progression methods. The garbling can also be determined via a very late check during post-processing, or an advanced type system could attempt to refuse it even statically.)

7. Access methods

For several typical annotation concerns, rewrite rules are predominantly unaware of annotations. For example, access restrictions should definitely be enforced without the programmer's intervention. Also, layout should be preserved largely for free. There are however concerns which require certain rewrite rules to be aware of annotations. In fact, a good example is fine tuning of comment preservation, where comments need to be moved from the input to the output on the basis of arguments that are specific to the transformation rules at hand.

Annotation-aware rewrite rules can *retrieve* annotations from the input and *establish* annotations in the output. To this end, the rewrite rules employ *access methods*, which reside in the 'annotation library' just as the progression methods. In general, access methods (and progression methods alike) can walk downwards (and maybe even upwards) the tree. So a method can basically reach all nodes in the parse tree and their annotations regardless of the current window.

Here is a (simplified) example that illustrates annotation-aware rewrite rules:

$$a(X_1, c{:}t_c(X_2, X_3)) \rightarrow a(X_1, f{:}t_f(X_3, X_2))$$

retrieve $comments = t_c.getComments()$
establish $t_f.putComments(comments)$

That is, we retrieve the comments from the vanishing function symbol c, and we establish the comment annotation for the introduced function symbol f. (Note that the vanishing comment annotations are not placed right at t_c but the access method $getComments$ will need to traverse.) As we can see, annotations that are retrieved from the input tree can be used in the method invocations that establish annotations in the output tree. In principle, retrieved annotations can also be used in constructing the output tree itself.

8. Related work

Annotations At a technical level, our approach to processing annotated parse trees exhibits one original ingredient: the provision of progression methods for computing annotations without a contribution from rewrite rules. The overall ability to annotate parse trees, abstract syntax trees or other data is widely established. For example, TXL [6] provides a so-called *attribute* mechanism, and ATerms [1] as used in the ASF+SDF Meta-Environment [2] and elsewhere literally reads as *annotated terms*. Also, XML's distinction of elements and *attributes* provides a simple annotation mechanism; simple because attributes are basically strings. At a more abstract level, our approach emphasises separation of concerns. A similar goal is addressed by a kind of annotation approach in [19]. Here, source text is *factored* according to different kinds of concerns such as the

ordinary code factor vs. the COPY prefactor and the COPY postfactor. Functionality can now observe all these factors and process the factors accordingly.

Layout preservation The specific concern of layout preservation has been addressed by a number of rewriting-based approaches. In the most simple case, no extension of the rewriting machinery is assumed [21], which implies some code tangling as discussed in Sec. 2. A more integrated approach is the one in [23] where overlays for term matching and building are used. While the primary term constructors contain layout positions, these overlays do not. A constant (normally a whitespace) is assumed for building a pattern in a rewrite rule. 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. All these approaches have in common that they rely on *extra branches* in a parse tree as opposed to our *extra annotation nodes*. 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 annotation concerns, and not just a specific form of layout preservation. Our work shows that layout preservation is just an instance of a more general problem: the annotation of parse trees (or similar data structures) with data needed for concerns that add to the syntactical structure. The contribution of our approach is that separate concerns can be effectively defined because annotation types, progression methods, and access methods are all programmer-definable.

Other 'separation of concerns' techniques There is another extension of rewriting which is related to our work: *origin tracking* [8]. Origins are relations between subterms of intermediate terms that occur during rewriting. A concern like change logging can obviously be handled using these relations. Origin tracking differs from our approach in so far that it does not involve any additional rewriting actions nor programmer-definable elements except for the ability to operate on origins *after* the normal form of a given term has been reached. Another, very general approach to the extension of the parse-tree format and to the corresponding adaptation of preexisting functionality is *meta-programming* as used for purposes similar to ours in [13, 18]. The later reference specifically demonstrates a meta-programming approach to layout preservation.

Attribute grammars vs. rewriting Our approach is related to attribute grammars [12] (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. 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. 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 [10]. 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 [9].

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 say that the AG describes 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.* 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 a systematic application of some rewrite rules all over the tree.

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 [22, 5, 20] (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 useful in practical applications of rewriting, in particular in the context of the implementation of transformations for software re-engineering. Our approach suggests to open up rewrite engines for performing actions on annotated trees in parallel with ordinary tree matching, building, and normalisation. This is in contrast to a rewrite engine that anticipates specific concerns. Our approach disentangles ordinary rewriting and annotation concerns. In particular, rewrite rules do not need to catch annotations when matching trees, neither do they need to supply annotations when building trees. With access methods, rewrite rules retrieve and establish annotations if this is necessary. With progression methods, rewrite steps are complemented so that required annotations are computed according to a given scheme. In our ongoing work, we aim at seamless support for the approach in language processing technology. To this end, a few conceptual issues deserve further research, especially typing, efficient use, and composition of progression methods.

Acknowledgement We are grateful for discussions with Paul Klint, Wolfgang Lohmann, Günter Riedewald, and Jurgen Vinju. We are also very grateful for the feedback on an earlier version of the paper, which was presented at the German GI-AOSD'03 workshop in Essen, in March 2003. Finally, thanks to the three enthusiastic, anonymous reviewers at SCAM'03.

References

[1] M. v. d. Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software – Practice and Experience*, 30(3):259–291, Mar. 2000.

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

[3] M. v. d. Brand, M. Sellink, and C. Verhoef. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes. In M. Sellink, editor, *Proc. Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer-Verlag, 1997.

[4] M. v. d. Brand and J. Vinju. Rewriting with Layout. In N. Derschowitz and C. Kirchner, editors, *Proc. Workshop on Rule-Based Programming (RULE'00)*, Sept. 2000.

[5] F. Burton and R. Cameron. Pattern Matching with Abstract Data Types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[6] J. Cordy. The TXL Programming Language, Mar. 2003. Version 10.3, `http://www.txl.ca/`.

[7] J. Cordy, K. Schneider, T. Dean, and A. Malton. HSML: Design Directed Source Code Hot Spots. In *Proc. International Workshop on Program Comprehension (IWPC'01)*. IEEE Press, May 2001.

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

[9] G. Hedin. An Overview of Door Attribute Grammars. In P. Fritzon, editor, *Proc. Compiler Construction (CC'94)*, volume 786 of *LNCS*, pages 31–51. Springer-Verlag, 1994.

[10] U. Kastens and W. Waite. Modularity and reusability in attribute grammars. *Acta Informatica 31*, pages 601–627, 1994.

[11] S. Klusener, R. Lämmel, and C. Verhoef. Architectural Modifications to Deployed Software. Submitted for publication, June 2002.

[12] D. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2:127–145, 1968. Corrections in 5:95-96, 1971.

[13] J. Kort and R. Lämmel. A Framework for Datatype Transformation. In B. Bryant and J. Saraiva, editors, *Proc. Language, Descriptions, Tools, and Applications (LDTA'03)*, volume 82 of *ENTCS*. Elsevier, Apr. 2003. 20 pages.

[14] J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In M. v. d. Brand and R. Lämmel, editors, *Proc. Language Descriptions, Tools, and Applications (LDTA'02)*, volume 65 of *ENTCS*. Elsevier Science, Apr. 2002.

[15] R. Lämmel. Beiträge zur Anpassung, Bewertung und Instrumentierung von Syntaxdefinitionen. In *Proc. Workshop Software Reengineering (WSR'00), Bad Honnef, Technischer Bericht Universität Koblenz*, May 2000.

[16] R. Lämmel and G. Riedewald. Prological Language Processing. In M. v. d. Brand and D. Parigot, editors, *Proc. Language Descriptions, Tools, and Applications (LDTA'01)*, volume 44 of *ENTCS*. Elsevier Science, Apr. 2001.

[17] R. Lämmel, E. Visser, and J. Visser. Strategic Programming Meets Adaptive Programming. In *Proc. Aspect-Oriented Software Development (AOSD'03)*, pages 168–177. ACM Press, 2003.

[18] W. Lohmann and G. Riedewald. Towards automatical migration of transformation rules after grammar extension. In *Proc. Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 30–39. IEEE Press, Mar. 2003.

[19] A. Malton, K. Schneider, J. Cordy, T. Dean, D. Cousineau, and J. Reynolds. Processing software source text in automated design recovery and transformation. In *Proc. International Workshop on Program Comprehension (IWPC'01)*. IEEE Press, May 2001.

[20] G. S. Novak Jr. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, 1995.

[21] M. Sellink and C. Verhoef. Scaffolding for Software Renovation. In J. Ebert and C. Verhoef, editors, *Proc. Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 161–172. IEEE Press, March 2000.

[22] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. Principles Of Programming Languages (POPL'87)*, pages 307–313. ACM Press, 1987.

[23] H. Westra. Configurable transformations for high-quality automatic program improvement. CobolX: a case study. Master's thesis, Utrecht University, February 2002.