Generating attribute grammar-based bidirectional transformations from rewrite rules

Pedro Martins João Saraiva
HASLab/INESC TEC & Univ. do Minho
Braga, Portugal
{prmartins, jas}@di.uminho.pt

João Paulo Fernandes
HASLab/INESC TEC & Univ. do Minho
RELEASE, Univ. da Beira Interior
Covilhã, Portugal
jpaulo@di.uminho.pt

Eric Van Wyk
Department of Computer Science and Engineering, Univ. of Minnesota
Minneapolis, Minnessota, USA
evw@cs.umn.edu

Abstract

Higher order attribute grammars provide a convenient means for specifying uni-directional transformations, but they provide no direct support for bidirectional transformations. In this paper we show how rewrite rules (with non-linear right hand sides) that specify a forward/get transformation can be inverted to specify a partial backward/put transformation. These inverted rewrite rules can then be extended with additional rules based on characteristics of the source language grammar and forward transformations to create, under certain circumstances, a total backward transformation. Finally, these rules are used to generate attribute grammar specifications implementing both transformations.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Data Types and Structures, Recursion; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages-Algebraic approaches to semantics; I.1.1 [*Symbolic and Algebraic Manipulation*]: Expressions and Their Representation

Keywords Attribute Grammars, Bidirectional Transformations, Rewrite Rules

1. Introduction

Attribute Grammars (AGs) [10] are a powerful mechanism to define complex transformations on tree structures. Originally, attribute grammars were mainly used to express the semantic analysis of a compiler, but they are now used to express complex multiple traversal algorithms [19], type system [15], pretty-printing algorithms [22], combinator languages [21], etc. Most of these AG-based algorithms rely on advanced extensions to the AG formalism that allow computations on graphs (reference attributes [8]) and dynamically evolving trees (higher order attributes [26]). All of these are efficiently executed by modern attribute grammar systems like, JastAdd [6], UUAG [25], Silver [24], and Kiama [17].

However, attribute grammars, and their modern extensions, only provide support for specifying unidirectional transformations, despite bidirectional transformations being common in AG applications. Bidirectional transformations are especially common between abstract/concrete syntax. For example, when reporting errors discovered on the abstract syntax we want error messages to refer to the original code, not a possible de-sugared version of it. Or when refactoring source code, programmers should be able to evolve the refactored code, and have the change propagated back to the original source code.

Another application is in semantic editors generated by AGs [11, 16, 18]. Such systems include a manually implemented bidirectional transformation engine to synchronise the abstract tree and its pretty printed representation displayed to users. This is a complex and specific bidirectional transformation that is implemented as two hand-written unidirectional transformations that must be manually synchronized when one of the transformations changes. This makes maintenance complex and error prone.

In this paper we show how bidirectional transformations can be modelled in an attribute grammar setting. We show how rewrite rules specifying a unidirectional forward (*get*) transformation can be inverted to specify a backward (*put*) transformation. These rules are used to generate AG specifications implementing both the *get* and *put* transformations. These specifications use a notion of origin tracking so that nodes in the target tree have access (via reference attributes) back to the node in the source tree from which they were created. This link back, when present, is used by put to produce the original source (sub) tree. The approach can be used even when the generated put is not total as it allows users to specify transformations that bring the target tree back into the domain of put or, as a last resort, specify a portion of the backward transformation manually. This is important for real languages in which (hopefully small) portions of the language may fall outside the scope of this approach.

1.1 Bidirectional Transformations

Bidirectional transformations are programs which express a transformation from one input to an output together with the reverse transformation, carrying any changes or modifications to the output, in a single specification. In the context of grammars, a bidirectional transformation represent a transformation from a phrase in one grammar to a phrase in the other, with the opposite direction automatically derived from the first transformation specification.

When applying the backward transformation to a modified tree, it is helpful to have access to the original tree to which the forward transformation was applied so that, at least, the unmodified parts map back to their original representation. For example, in a transformation $A \rightarrow B$, a bidirectionalization system defines the $B \rightarrow A$ transformation, which has to carry any upgrades applied to B back to a new A' which is as close as possible to the original A.

PEPM '14, January 20–21, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2619-3/14/01...$15.00.
http://dx.doi.org/10.1145/2543728.2543745

These transformations can be coded in AGs. Of special interest are tree-based structures such as the ones generated by concrete and abstract grammars. The problem with these transformations is that both the forward and the backward transformations need to be implemented by hand.

In AGs, we can use annotations (a type of attribute computed when the tree is constructed) and reference attributes so that each node in the generated abstract tree has a link back to the node in the concrete tree that constructed it. This is quite useful in our approach for ensuring that the backward transformation can transform unmodified sub-trees back to there original form in the source. Changes on the abstract side will cause some of these links to be discarded as new link-free tree nodes are created. The presence of a link back indicates that a tree rooted at that node was created during the original transformation from concrete to abstract trees. Furthermore, the put transformation knows what tree created that tree being "put back" and can use the link back as its result, making the transformation very simple and producing as result the original concrete tree.

The main problem is that coding this all by hand in AGs is quite tedious and error prone, and dealing with presence and absence of links back is an extra task that has to be always performed. In this work we will see how we automatically generate attribute grammar specifications that are much more complex that we have typically wanted to write and thus have rarely done.

Another interesting point to consider are the possible mappings back to the concrete grammar. As we will see, sometimes each expression in the abstract tree needs to have rules for more than one transformation back, since it is often the case that multiple concrete syntax nonterminals will map to a single terminal in the abstract grammar. The automatic generation of these mappings is another feature of our system.

1.2 Outline of solution and challenges

In our approach we will specify transformations as rewrite rules, and define an inversion technique for such rules to specify backward transformations. These are straightforward for applications where both grammars are isomorphic and the rules define a bijective transformation since there is a one-to-one mapping between productions of the grammars. Our challenge is to propose an approach that works on real, non-isomorphic, grammars that have a different number of symbols and productions and different grammatical structure.

To make this possible, we extend transformations with links from that target syntax tree back to the source so that unchanged trees know what they were transformed from and can transform directly back to. Furthermore, we want a solution that works on many types of transformations and grammars, but can be used in a system that gracefully degrades when a component of that transformation cannot be handled automatically by the inversion of rewrite rules. Sometimes the target tree is manipulated so that if falls out of the ranges of the get transformation. We thus need an allowance for repairing trees to bring them back into this range. Other times the get transformation cannot be specified as rewrite rules and we would like to use the system still, if only on those parts of the language for which it will work. In these cases hand-coded techniques can still be applied.

1.3 Contributions

As contributions, in this paper we:

- Use an algebraic approach to define invertible rewrite rules that specify bidirectional transformations between grammars.
- Define an automation of strategies to maintain links to the source tree. We also explain how attribute specifications that implement these features can be automatically produced.
- Support non-linear rewrite rules to handle complex transformation, and define automatic techniques both to invert these rules and to generate attribute specifications from them.
- Formalize dynamic analyses of trees in the form of attribute specifications that automatically detect if certain trees or sub-trees need special refactoring strategies to fit into the scope of the transformations. The point is to allow some manual intervention for the, hopefully small, portions of the transformations that cannot be generated completely automatically.
- Provide a prototype that implements all the features and specifications described in this paper.

In this paper section 2 provides some background. Section 3 explains how rewrite rules can define bidirectional transformations, including links back to the source. In Section 4 we show how non-linear rules and tree refactoring are supported. Section 5 discusses related work and Section 6 concludes.

2. Background

We start with a little background needed to understand the paper.

Algebraical Background: We use an algebraic approach and benefit from existing techniques from Chirica et al [2] and Courcelle and Franchi-Zannettacci [3] in defining AGs algebraically.

We start by defining an operator scheme: $\Sigma = \langle S, F, \sigma \rangle$ where S is a set of sorts (sort names), F is a set of operator or function names and σ maps F to $S^* \times S$. For grammars, sorts correspond to nonterminals and terminals, operators correspond to production names, and signatures in σ correspond to productions. Constants are treated as nullary operators. A Σ-algebra: A_σ is defined as:

- $\{A_s\}_{s \in S}$ - an S-indexed family of sets, called *carrier sets*
- $\{f^A : A_{s_1} \times A_{s_2} \times ... \times A_{s_n} \to A_s \mid f \in F, \sigma(f) = S_1 \times S_2 \times ... \times S_n \to S\}$. For each function name $f \in F$ there is a function f^A over the appropriate carrier sets in $\{A_s\}_{s \in S}$ as indicated by the signature of f, $\sigma(f)$.

Word algebras, over variables, specify (ground) terms or patterns, if variables are used, and are denoted $W_\Sigma(V)$ for variables V. An example of a word is "$plus(mul(a, b), c)$", where 'a', 'b' and 'c' are variables. We will find a need to order patterns (words with variables) from least specific to most specific. We use a standard notion of specificity in that one word is more specific than another if the set of ground terms created from all instantiations is a subset of such ground terms for another pattern.

In this paper, the terms $term$, $word$ and $tree$; the terms $production$, $operator$ and $constructor$ and the terms $type$, $sort$ and $nonterminal$ have the same meaning and are interchangeably.

Attribute Grammars: In attribute grammars, equations define semantic-valued *attributes* on syntax tree nodes [10]. Due to space limitations AGs, and their extensions [8, 19, 24, 26] are explained as needed below.

3. Generating backward transformations from rewrite rules

In this section we will see how rewrite rules are used to define transformation specifications and how these can be automatically inverted to form a backward transformation.

Source language: $\Sigma^E = \langle S^E, F^E, \sigma^E \rangle$ where:

- $S^E = \{E, T, F, \mathit{digits}, `+', `-', `*', `/', `(', `)', \mathit{String}\}$,
- $F^E = \{\mathit{add}, \mathit{sub}, \mathit{et}, \mathit{mul}, \mathit{div}, \mathit{tf}, \mathit{nest}, \mathit{const}, \mathit{digits}, \mathit{neg}, `+', `-', `*', `/', `(', `)', \mathit{String}\}$,
- $\sigma^E(\mathit{add}) = E\ `+'\ T \to E$,
  $\sigma^E(\mathit{sub}) = E\ `-'\ T \to E$,
  $\sigma^E(\mathit{et}) = T \to E$,
  $\sigma^E(\mathit{mul}) = T\ `*'\ F \to T$,
  $\sigma^E(\mathit{div}) = T\ `/'\ F \to T$,
  $\sigma^E(\mathit{tf}) = F \to T$,
  $\sigma^E(\mathit{nest}) = `('\ E\ `)' \to F$,
  $\sigma^E(\mathit{neg}) = `-'\ F \to F$,
  $\sigma^E(\mathit{const}) = \mathit{digits} \to F$,
  $\sigma^E(\mathit{digits}) = \mathit{String} \to \mathit{digits}$,
  $\sigma^E(`+') = \epsilon \to `+'$,
  $\sigma^E(`-') = \epsilon \to `-'$, ...

Target Language: $\Sigma^A = \langle S^A, F^A, \sigma^A \rangle$ where

- $S^A = \{A, \mathit{String}\}$
- $F^A = \{\mathit{plus}, \mathit{minus}, \mathit{times}, \mathit{divide}, \mathit{constant}\}$
- $\sigma^A(\mathit{plus}) = A\ A \to A$,
  $\sigma^A(\mathit{minus}) = A\ A \to A$,
  $\sigma^A(\mathit{times}) = A\ A \to A$,
  $\sigma^A(\mathit{divide}) = A\ A \to A$,
  $\sigma^A(\mathit{constant}) = \mathit{String} \to A$

Figure 1: Concrete and abstract syntax of arithmetic expressions.

3.1 Source and target languages

Our first example is given in Figure 1 and shows the operator scheme Σ^E for the source language E and Σ^A for the target language A. Some readers will be more familiar with the BNF notation for context free grammars, which is similar to algebras and can be easily translated into this algebraic setting. Nonterminal and terminal symbols become sorts. The sorts E, T, F in S^E correspond to the nonterminals commonly used in this example. The sort digits represents an integer literal terminal, and the operator and punctuation symbols are given sort names by quoting the symbol. For example, $`+'$ corresponds to the terminal symbol for the addition symbol. Strings are also used and play the role of lexemes on scanned tokens; thus we have the sort String.

The productions in a grammar correspond to operators, in this example: $\mathit{add}, \mathit{sub}, \mathit{et}, \mathit{mul}, \mathit{div}, \mathit{tf}, \mathit{nest},\ \mathit{neg}, \mathit{const} \in F^E$ for the concrete syntax. The signature of each of these operators is given by σ^E and is written "backwards" from how they appear in BNF. For example, the operator add is a ternary operator taking values of sort E, $`+'$, and $T$ and creating values of type $E$, as denoted by $\sigma^E(\mathit{add}) = E\ `+'\ T \to E$.

There is also a single operator for each terminal symbol. If the regular expression that would be associated with a terminal in its scanner specification is constant, then this operator is nullary. If it is not constant but identifies a pattern for, say, variable names or integer constants, then we make the signature unary with String being the single argument. We will refer to nullary terminal operators as constant, and unary terminal operators as non-constant. To avoid too much notational clutter we will overload sort and operator names for those corresponding to terminal symbols.

- The sort map: $sm^{get} :: F^E \to 2^{F^A}$
  $sm^{get}(E) = \{A\},\ \ sm^{get}(T) = \{A\},\ \ sm^{get}(F) = \{A\}$
- The rewrite rules rw^{get}:

$$
\begin{array}{l}
get_A^E(\mathit{add}(l, `+', r)) \to \mathit{plus}(get_A^E(l), get_A^T(r))\\
get_A^E(\mathit{sub}(l, op, r)) \to \mathit{minus}(get_A^E(l), get_A^T(r))\\
get_A^E(\mathit{et}(t)) \to get_A^T(t)\\
get_A^T(\mathit{mul}(l, `*', r)) \to \mathit{times}(get_A^T(l), get_A^F(r))\\
get_A^T(\mathit{div}(l, `/', r)) \to \mathit{divide}(get_A^T(l), get_A^F(r))\\
get_A^T(\mathit{tf}(f)) \to get_A^F(f)\\
get_A^F(\mathit{neg}(`-', r)) \to \mathit{minus}(\mathit{constant}(``0"), get_A^F(r))\\
get_A^F(\mathit{nest}(`(', e, `)')) \to get_A^E(e)\\
get_A^F(\mathit{const}(\mathit{digits}(d)) \to \mathit{constant}(d)
\end{array}
$$

Figure 2: User provided forward transformation specification.

Trees in this language are written as terms or words from the corresponding word algebra, parametrized by a set of strings representing lexemes. This algebra is technically denoted $W_\Sigma(\mathit{String})$ but we omit String below. We overload String to denote the sort, as in Figure 1, and here to denote the carrier set of strings.

3.2 Specifying the forward transformation

As in most approaches to bidirectional transformation, the forward transformation is provided and used to generate the backward transformation. Here we describe the structure of the forward specifications used in our approach and provide the forward transformation specification from Σ^E to Σ^A, which is shown in Figure 2.

In defining the forward transformation, the first part of the specification is the *sort-map*, which in our approach will be generalized so that the range of the sort map is a set of sorts in the target. In our first example, this maps all of the source sorts of expressions (E, T, and F) to the single sort for expressions in the target/abstract scheme A.

The patterns used in the rewrite rules to specify the translation from the source to the target are not merely terms from the (word algebra of the) source or target language (extended with variables). We create additional operators, based on the sort map, whose signatures include sorts from both the source language and the target language. From a sort map sm, we create additional operators:

$$\{get_Y^X | sm(X) = Y\}$$

indicating that the forward (get) transformation maps an X in the source to a Y in the target. The signatures for such operators are as expected:

$$\sigma^{get}(get_Y^X) = Y \to X, \forall X \in S^S, Y \in sm(X)\ .$$

The left and right hand sides of the rewrite rules are then words in a word algebra for the operator scheme that include both the source and target operator schemes and these attribute-like operators. Left hand side and right hand side patterns are words in $W_{\Sigma^{get}}(V)$ for a sort-indexed set of variable names V. Both the left and right hand side are terms of the same target language sort. Note that we do not have rules for sorts corresponding to terminal symbols; they have no translation in the target.

Restrictions on forward transformation specifications: We place a number of restrictions on forward transformation specifications from $\Sigma^S = \langle S^S, F^S, \sigma^S \rangle$ to $\Sigma^T = \langle S^T, F^T, \sigma^T \rangle$ to ensure that a backward transformation can be generated. Some of these restrictions are removed in later sections. We assume only type correct words are used here and in the remainder of the paper.

1. First, $|sm(X)| \leq 1, X \in S^S$. Each sort in the source maps to one or zero items in the target.
2. Second, words on the left hand side of a rewrite rule must have the form $get^X_Y(p(v_1, ..., v_n))$ for some $p \in F^S$ in which $v_1, ..., v_n$ are variables or are terms that do not contain variables. Furthermore, all variables on the left hand side must either (i) appear on the right hand side, or (ii) be of a sort that contains only one value, for example the sorts of so-called *constant* terminal symbols.
3. When viewed as patterns, the words on the left hand side of the rules must not be overlapping, that is there can be no substitution of their variables that results in the same word.
4. We also need to ensure that the forward transformation specifies a total function from Σ^S to Σ^T.

   Definition 1. *A transformation specification from source $\Sigma^S = \langle S^S, F^S, \sigma^S \rangle$ to target $\Sigma^T = \langle S^T, F^T, \sigma^T \rangle$ is* total *if for each $X \in S^S$ and $Y \in sm(X)$ there exists a rule with the left hand side of the form $get^X_Y(p(v_1, ..., v_n))$ for each $p \in F^S$ in which $v_1, ..., v_n$ are variables.*

   In terms of attribute grammars this is the definition of an attribute grammar that passes the closure test [10].
5. Words on the right hand side are also restricted. For a rule with the left hand side
   $$get^X_Y(p(v_1, ..., v_n))$$
   the right hand side must be a word in which for any sub-word of the form $get^{X'}_{Y'}(w)$, the word w must be a variable. We will lift this restriction in a later section.

Generating attribute grammar equations: Since our aim is to implement both the forward, and generated backward, transformations in attribute grammars we need to convert the rewrite rules to attribute grammar equations.

Generating attribute grammar equations from rules of this type specified above is quite straightforward. For example, the rule

$$get^E_A(add(l, `+', r)) \to plus(get^E_A(l), get^T_A(r))$$

is expressed as the following attribute grammar equation on the add production that has the signature $e :: E ::= l :: E$ ' + ' $r :: T$:

$$e.get^E_A = plus(\ l.get^E_A, r.get^T_A)$$

In later examples we will see that rewrite rules can be more complex and thus the translation to attribute grammar code is less direct.

3.3 Generating the backward transformation

In this section we describe the process for inverting the forward transformation to generate the backward one.

Inverting the sort map and rewrite rules: The first step is to invert the sort map. In our example this inversion leads to a sort map sm^{put} that maps the abstract sort A back to three concrete sorts E, T, and F. The inverted sort map now maps target sorts to multiple source sorts. Thus, we really are defining 3 put transformations: putting an A back to an E, back to a T, and back to an F. This is the basis for the put operators that are analogous to the get operators seen above.

The second step is to invert the rewrite rules. The result of this process for the rules in Figure 2 produces the rules in Figure 3.

- The sort map: $sm^{put} :: F^A \to 2^{F^E}$
  $sm^{put}(A) = \{E, T, F\}$
- The rewrite rules rw^{put}

$$\begin{array}{l}
put^A_E(plus(l, r)) \to add(put^A_E(l), `+', put^A_T(r)) \\
put^A_E(minus(l, r)) \to sub(put^A_E(r), `-', put^A_T(r)) \\
put^A_E(minus(constant(``0"), r)) \to neg(`-', put^A_F(r)) \\
put^A_E(t) \to et(put^A_T(t)) \\
put^A_T(times(l, r)) \to mul(put^A_T(l), `*', put^A_F(r)) \\
put^A_T(divide(l, r)) \to div(put^A_T(l), `/', put^A_F(r)) \\
put^A_T(f) \to tf(put^A_F(f)) \\
put^A_F(e) \to nest(`(', put^A_E(e), `)') \\
put^A_F(constant(d)) \to const(digits(d))
\end{array}$$

Figure 3: Direct inversion of forward transformation specification.

Given the restriction on the forward transformation, this process is relatively straightforward. A rule of the form

$$get^X_Y(w(v_1, ..., v_n)) \to w'(get^{X_1}_{Y_1}(v_1), ..., get^{X_1}_{Y_1}(v_n))$$

where v_i is of sort X_i and $sm(X_i) = \{Y_i\}$ is inverted to form the rule

$$put^Y_X(w'(v'_1, ...v'_n)) \to w(put^{Y_1}_{X_1}(v'_1), ..., put^{Y_n}_{X_n}(v'_n))$$

in which the variables v'_i are of sort Y_i. This can be seen in the inverted rules in Figure 3.

Extending the rules: Consider a transformation in an abstract syntax that creates the subtree $times(\_, plus(\_, \_)$. While the second argument of *mul* is of sort *F*, *plus* maps most directly back to an *E*. Thus the backward transformation must create a source term of type F from term $plus(\_, \_)$.

The key to solve this problem lies in the rules with a right hand side of the form $put^Y_X(v) \to w(put^Y_{X'}(v))$, where w is a word containing the sub-word $put^Y_{X'}(v)$ which holds the only variable, namely v. Such a rule shows how to transform any term of type X' in the source language to one of type X in the source language. For example, the rule $put^A_F(e) \to nest(`(', put^A_E(e), `)')$ in Figure 3 shows that a term of type E can be converted to one of type F by wrapping it in parenthesis, that is, in the term $nest(`(', \_, `)')$.

We specialize the inverted rewrite rules of this form so that their left hand sides are of the form $get^X_Y(p(v_1, ..., v_n))$ for $p \in F^A$ in which $v_1, ..., v_n$ are variables of the appropriate type:

1. If
   (a) $\exists X \in S^S$ and $Y \in sm(X)$ such that there does not exists a rewrite rule whose left hand side has the form $gp^X_Y(p(v_1, ..., v_n))$ for some $p \in F^S$ such that return type of p is X (For some α, $\sigma^\Sigma(p) = \alpha \to X$) and for some variables $v_1, ..., v_n$, and
   (b) there exists a rule of the form $gp^X_Y(t) \to w(gp^X_{Y'}(t))$

   then add the rule $gp^X_Y(p(v_1, ..., v_n)) \to w(gp^X_{Y'}(p(v_1, ..., v_n)))$
2. Repeat step 1 until no more rules can be added.

For example, the rule $put^A_F(e) \to nest(`(', put^A_E(e), `)')$ in Figure 3 shows that a term of type E can be converted to one of type F by wrapping it in parenthesis, that is, in the term $nest(`(', \_, `)')$.

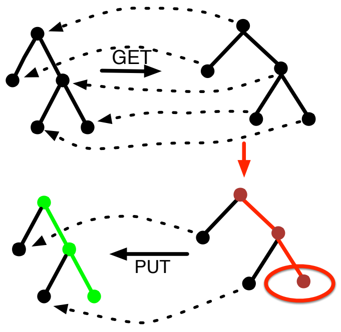

Figure 4: Using links back.

From the original rules we would add the following rule:

$$put_F^A(plus(l,r)) \to nest(`(', put_E^A(plus(l,r)), `)') \ .$$

and then repeat this process until we have as much extended rules as possible. The extended, inverted rewrite rules can now be checked for totality with the original definition of totality 1.

Generating attribute grammar equations: From these extended set of rules, we generate AG equations as described before, resulting in three equations on productions such as *plus*: one for put_E^A, put_T^A, and put_F^A. Notably, the generated *put*transformation does not add any unnecessary parenthesis, a common problem with simple string-based printing mechanisms.

3.4 Links back: making use of the original source term

The perceptive reader might have noticed there is a specific production, *neg*, for which generating attribute equations presents a new challenge. The issue is that the inversion of that rule and the rule for *sub* creates backward transformation rules with overlapping left-hand sides, as seen in Figure 3. To address this, the *get* transformation adds *links* on generated abstract (sub) trees that refer back to the respective concrete (sub) tree that generated them.

Figure 4 shows this situation diagrammatically: the source tree and the target with links back is shown on top. In transforming the target tree, links back do not exist on newly constructed nodes as indicated by the lack of arrows back to the source tree. The portion in the oval (in red) was newly created, as well as the nodes on the spine to the root of the tree.

To be concrete, consider the phrase

$$sub(et(tf(const(digits(``0")))), `-', x)$$

that became

$$minus(constant(``0"), x)$$

in the target. If the original (sub) tree in the abstract tree was maintained during a transformation then it has a link back to the original tree of sort E and can return that when queried for its *put* attribute put_E^A. This ensures that we can recover the original source tree in the *put* transformation, when the *put* transformation is mapping it back to its original type, here E. In case that the link back is of sort F, the *can-be* rules can be used in a manner similar as they were above. This can be seen in the generated attribute equation for *minus* for put_E^A below:

$$\begin{array}{l} minus: \quad s :: A ::= l :: A \quad r :: A \\ \quad s.put_E^A = \textbf{case } s.link \textbf{ of} \\ \qquad\qquad link_E(e) \to e \\ \qquad\qquad link_F(f) \to et(tf(f)) \\ \qquad\qquad \bot \to sub(\ l.put_E^A,\ r.put_E^A\ ) \end{array}$$

Source language: $\Sigma^C = \langle S^C, F^C, \sigma^C \rangle$ where:

- $S^C = \{Root_C, Decls_C, Decl_C, Vars_C, Type_c, `,', `;', String\}$,
- $F^C = \{root_C, consDecl_C, nilDecl_C, multiDecl_C, consVar_C, oneVar_C, intType_C, floatType_C, arrayType_C, `,', `;', String\}$
- $\sigma^C(root_C) = Decls_C \to Root_C$,
  $\sigma^C(consDecl_C) = Decl_C\ Decls_C \to Decls_C$,
  $\sigma^C(nilDecl_C) = \epsilon \to Decls_C$,
  $\sigma^C(multiDecl_C) = Type_C\ Vars_C\ `;' \to Decl_C$,
  $\sigma^C(consVar_C) = String\ `,'\ Vars_C \to Vars_C$,
  $\sigma^C(oneVar_C) = String \to Vars_C$,
  $\sigma^C(intType_C) = `int' \to Type_C$,
  $\sigma^C(floatType_C) = `float' \to Type_C$,
  $\sigma^C(arrayType_C) = `array'\ Type_C \to Type_C$

Target: $\Sigma^A = \langle S^A, F^A, \sigma^A \rangle$ where

- $S^A = \{Root_A, Decl_A, Type_A, String\}$
- $F^A = \{root_A, seqDecl_A, skipDecl_A, intType_A, floatType_A, arrayType_A, String\}$
- $\sigma^A(root_A) = Decl_A \to Root_A$,
  $\sigma^A(seqDecl_A) = Decl_A\ Decl_A \to Decl_A$,
  $\sigma^A(skipDecl_A) = \epsilon \to Decl_A$,
  $\sigma^A(varDecl_A) = Type_A\ String \to Decl_A$,
  $\sigma^A(intType_A) = \epsilon \to Type_A$,
  $\sigma^A(floatType_A) = \epsilon \to Type_A$,
  $\sigma^A(arrayType_A) = Type_A \to Type_A$

Figure 5: Concrete and abstract syntax for variable declarations.

One important remark is this process, maintenance and usage of the links back (and the *canbe* relation) is completely automatic and requires no extra coding.

3.5 Allowing overlapping rewrite rules

In order to generate equations we need to sort the rewrite rules based on the specificity of the left hand side and use these left hand side in a **case**-expression to select the most specific one.

On the transformation that have *minus* as domain (Figure. 3), we sort the rewrite rules so that the most precise one is used before the more general ones: the transformation tries to apply $put_F^A(minus(constant(``0"), f))$ first, because it is more specific.

The techniques described in this section create a setting where we can describe complex transformations between grammars with various nonterminals and produce, from these rules, attribute specifications for the transformation in both directions.

4. Supporting non-linear, compound rules and partial transformations

In this section we extend the process described above to apply to non-linear, compound rewrite rules. We also describe a means for handling two situations in which the generated backward transformation is partial. The first is when some hand-written manipulation of the target tree can move it back into the domain of the backward transformation; the second is fall-back case in which some portion of the backward transformation must be written manually.

- The sort map: $sm^{get} :: F^C \to 2^{F^A}$
  $sm^{get}(Root_C) = \{Root_A\}$,
  $sm^{get}(Decl_A) = \{Decls_C,\ Decl_C\}$,
  $sm^{get}(Type_A) = \{Type_C\}$
- The rewrite rules rw^{get}:

$$
\begin{array}{l}
get^{Root_C}_{Root_A}(root_C(c)) \to root_A(get^{Decls_C}_{Decl_A}(c)) \\
get^{Decl_C}_{Decls_A}(consDecl_C(d, rest)) \\
\qquad \to seqDecl_A(get^{Decl_C}_{Decl_A}(d), get^{Decls_C}_{Decl_A}(rest)) \\
get^{Decls_C}_{Decl_A}(nilDecl_C())) \to skipDecl_A() \\
get^{Decl_C}_{Decl_A}(multiDecl_C(t, oneVar_C(v),';')) \\
\qquad \to varDecl(get^{Type_C}_{Type_A}(t), v) \\
get^{Decl_C}_{Decl_A}(multiDecl_C(t, consVar_c(v,',', rest),';')) \\
\qquad \to seqDecl(varDecl(get^{Type_C}_{Type_A}(t), v), \\
\qquad\qquad\qquad get^{Decl_C}_{Decl_A}(multiDecl_C(t, rest,';'))) \\
get^{Type_C}_{Type_A}(intType_C()) \to intType_A() \\
get^{Type_C}_{Type_A}(floatType_C()) \to floatType_A() \\
get^{Type_C}_{Type_A}(arrayType_C()) \to arrayType_A()
\end{array}
$$

Figure 6: User provided forward transformation specification for declarations.

4.1 Non-linear, Compound Rule Specifications

We start by presenting in Figure 5 a new pair of concrete and abstract algebras/grammars. The concrete syntax allows sequences of declarations of the form "$int\ x,\ y,\ z;$" while the abstract requires simpler declarations of just one variable, and thus this example becomes "$int\ x; int\ y; int\ z$" in the abstract.

To support transformations such as this, we allow the right hand side of rewrite rules to have attribute operators (get^X_Y) that contain a term (a tree) instead of just a variable. Such rules are called *compound*. In particular, for the grammars defined in Figure 5, consider the two rules for the production $multiDecl_C$ (also shown in Figure 6):

- $get^{Decl_C}_{Decl_A}(multiDecl_C(t,\ oneVar_C(v, `,')))$
  $\to varDecl(get^{Type_C}_{Type_A}(t), v)$
- $get^{Decl_C}_{Decl_A}(multiDecl_C(t,\ consVar_C(v, `,', rest)))$
  $\to seqDecl(\ varDecl(get^{Type}_{Type_C}(t), v),$
  $get^{Decl_C}_{Decl_A}(multiDecl_C(t, rest, `;'))\ )$

The first rule is similar to the ones we have seen in previous sections. In the more interesting second rule, the right hand side creates a new term/tree in the concrete syntax ($multiDecl_C(t, rest, `;')$) on which we recursively apply the forward transformation $get^{Decl_C}_{Decl_A}$.

To generate the forward transformation AG equations for the first rule we use the strategy presented in Section 3.3. For the second rule, the same process now defines a new concrete tree and then accesses the $get^{Decl_C}_{Decl_A}$ attribute on that "locally" created tree. This process, described in general below, creates the following equation for the $get^{Decl_C}_{Decl_A}$ on the $multiDecl_c$ production:

$$
\begin{array}{l}
multiDecl_c :\quad d :: Decl_c ::= t :: Type_C\ vars :: Vars_C\ `;' \\
\quad d.get^{Decl_C}_{Decl_A} = \textbf{case}\ d\ \textbf{of} \\
\qquad multiDecl_C(t, oneVar_C(v, \_), \_) \to varDecl(t.get^{Type_c}_{Type_A}, v) \\
\qquad multiDecl_C(t,\ consVar_C(v, \_, rest), \_) \\
\qquad\quad \to seqDecl(\ varDecl(t.get^{Type_C}_{Type_A}, v), \\
\qquad\qquad\qquad (multiDecl_C(t, rest,';')).get^{Decl_C}_{Decl_A}\quad)
\end{array}
$$

Inverting the rewrite rules: Inverting the rules for $multiDecl_C$ creates non-linear rewrite rules with side conditions. The inverted rules for $multiDecl_C$ are shown below:

- $put^{Decl_A}_{Decl_C}(varDecl_A(t, v))$
  $\to multiDecl_C(put^{Type_A}_{Type_C}(t),\ oneVar_C(v), `;')$
- $put^{Decl_A}_{Decl_C}(\ seqDecl_A(varDecl(t, v), right)\ )$
  $\to multiDecl_C(put^{Type_A}_{Type_C}(t),\ consVar_C(v, `,', rest))$
  where $put^{Decl_A}_{Decl_C}(right) = multiDecl_C(put^{Type_A}_{Type_C}(t), rest, `;')$

Previously, on forward transformation rules, (i) variables on the left were wrapped with a put operator on the right of the put rule, and (ii) variables on the right wrapped by a get operator became (unwrapped) variables on the left of the put rule. This can be seen in the first inverted rule above. For the second rule, we generalize this process so that in the forward transformation rules, a term trm on the right wrapped by a get operator becomes new variable ($right$ in the case above) on the left of the put rule and creates a side condition of the form $put(right) = trm'$, where trm' is the result of applying this process to trm. Note that this generalized process wraps variables of sorts in the target with a put operator, but not those with a sort in the source. In the above example t is wrapped with $put^{Type_A}_{Type_C}$ but $rest$ is not. Note that applying this extended process to the first $multiDecl_C$ rule results in one that directly simplifies to the one shown above.

Generating attribute grammar equations: Generating the attribute equations for compound, non-linear rules with side conditions used either in the forward or generated backward transformation requires extending process described in the previous section, where the case-expressions are used as components in the attribute equations when no links back to the source applied and thus fit into the equations as previously specified.

The pattern p in the left hand side of the rule becomes the pattern on the left of the case clause. The existence of side conditions induce a nested case on newly created variables (see above) on which the appropriate put attribute has been applied. This can be seen in the equation, shown in Figure 7, for the second $multiDecl_C$ rule. The nested case checks that the target tree ($right$) could be generated, by the get transformation, from the source tree in the compound part of the term. In our example we check if $right.put^{Decl_A}_{Decl_C}$ matches the tree nested under a put: $multiDecl_C(t, rest,';')$ under $put^{Decl_A}_{Decl_C}$ in the compound rule above.

For the non-linear aspect we could generate case clauses of the form $pattern \mid check \ \to\ expr$ in which $check$ is a Boolean expression that uses variables matched in $pattern$ (here to check that they are equal) that must evaluate to $true$ for the clause to be used. In our case we are generating attribute grammar for Silver, an AG system that does not support such case clause and thus we extract them into a separate *if-then-else* expressions. This is seen in the partial equation for $put^{Decl_A}_{Decl_C}$ in Figure 7.

4.2 Tree Repairs

In this example, the sequences of declarations in the source and target take different forms. In the source, it is a list formed by traditional *cons* and *nil* operators. But in the target, a more general tree structure is allowed using *seq* and *skip* (empty) operators. Using the techniques described above, the range of the get transformation is not the full range of valid sequences of declarations in the

$$
\begin{array}{l}
\mathit{seqDecl}_A : d :: \mathit{Decl}_A ::= d_1 :: \mathit{Decl}_A \;\; d_2 :: \mathit{Decl}_A \\
d.\mathit{put}^{\mathit{Decl}_A}_{\mathit{Decl}_C} = \textbf{case } d \textbf{ of} \\
\qquad \mathit{seqDecl}_A(\mathit{varDecl}_A(t_1, v), \mathit{right}) \rightarrow \textbf{case } \mathit{right}.\mathit{put}^{\mathit{Decl}_A}_{\mathit{Decl}_C} \textbf{ of} \\
\qquad\qquad \mathit{multiDecl}_C(t_2, \mathit{rest}, \text{`;'}) \;\rightarrow\; \textbf{if } t_2.\mathit{put}^{\mathit{Type}_A}_{\mathit{Type}_C} == t_1 \textbf{ then } \mathit{multiDecl}_C(t_2, \mathit{consVarC}(v, \text{`,'}, \mathit{rest}), \text{`;'}) \\
\qquad\qquad\qquad \textbf{else } \mathit{error}(\text{``}\mathit{Failed\ to\ pattern\ match!}\text{''}) \\
\\
d.\mathit{needs\_repair}^{\mathit{Decl}_A}_{\mathit{Decl}_C} = \textbf{case } d \textbf{ of} \\
\qquad \mathit{seqDecl}_A(\mathit{varDecl}_A(t_1, v), \mathit{right}) \rightarrow \textbf{case } \mathit{right}.\mathit{put}^{\mathit{Decl}_A}_{\mathit{Decl}_C} \textbf{ of} \\
\qquad\qquad \mathit{multiDecl}_C(t_2, \mathit{rest}, \text{`;'}) \;\rightarrow\; \textbf{if } t_2.\mathit{put}^{\mathit{Type}_A}_{\mathit{Type}_C} == t_1 \textbf{ then } \mathit{false} \textbf{ else } \mathit{true}
\end{array}
$$

Figure 7: Sketch of attribute specification and the $\mathit{needs\_repair}$ attribute from non-linear, compound rewrite rules with side conditions.

target and thus the generated backward transformation is not total, it is only partial. Only sequences that have a list-like shape in the abstract can be mapped back to the concrete.

To see this, note that the inversion of the rule

$$
\begin{array}{l}
\mathit{get}^{\mathit{Decl}_A}_{\mathit{Decls}_C}(\mathit{consDecl}_C(d, \mathit{rest})) \\
\rightarrow \mathit{seqDecl}_A(\mathit{get}^{\mathit{Decl}_C}_{\mathit{Decl}_A}(d), \mathit{get}^{\mathit{Decls}_C}_{\mathit{Decl}_A}(\mathit{rest}))
\end{array}
$$

yields the following rule, with variables changed to be more appropriate for the abstract syntax, for the backward transformation:

$$\mathit{seqDecl}_A(d_1, d_2) \rightarrow \mathit{consDecl}_C(\mathit{get}^{\mathit{Decl}_A}_{\mathit{Decl}_C}(d_1), \mathit{get}^{\mathit{Decls}_A}_{\mathit{Decl}_C}(d_2))$$

Note that from an abstract production $\mathit{seqDecl}_A$, we need to get a Decl_C from the left child d_1 and a Decls_c from the right child d_2. If d_1 is of the form $\mathit{varDecl}(t, n)$ this is no problem since there will be a transformation rule from these back to a $\mathit{multiDecl}_C$, via one of the rules in Figure 6. But if d_1, after some transformation on the abstract tree has taken place, is a $\mathit{seqDecl}_A$ or a $\mathit{skipDecl}_A$ then this tree is not in the domain of the backward transformation.

However, in many cases like this, it is possible to *repair* such trees and convert them back to a list-like structure so that they are in the domain of the generated put. We extend these techniques to generated AG equations to detect if the target tree is in the domain of the put transformation: in this case, a $\mathit{needsRepair}^{\mathit{Decl}_A}_{\mathit{Decl}_C}$ attribute that is true on nodes of sort Decl_A that need to be repaired before transforming back to a Decl_C sort in the source.

Generating equations for a $\mathit{needsRepair}^{\mathit{Decl}_A}_{\mathit{Decl}_C}$ attribute is quite straightforward since is has the same structure as the equations for the corresponding attribute $\mathit{put}^{\mathit{Decl}_A}_{\mathit{Decl}_C}$, as can be seen in the second equation in Figure 7. Constructed trees are replaced by false and error-cases are replaces by true.

In such cases, the user must write attribute equations, $\mathit{repair}^{\mathit{Decl}_A}_{\mathit{Decl}_C}$ in this case, that convert the tree into a form that is in the domain of the put transformation. In this case an accumulating inherited attribute can be used to provide a Decl_A with the tail of the list that should follow it. The point being that such repairs can be made, but equations must be written by hand.

In generating the put attribute equations, the framework will then replace attribute accesses of the form $n.\mathit{put}^{\mathit{Decl}_A}_{\mathit{Decl}_C}$ with expressions that query the $\mathit{needsRepair}^{\mathit{Decl}_A}_{\mathit{Decl}_C}$ attribute: if it is true the tree is first repaired before performing the put transformation, otherwise the $\mathit{put}^{\mathit{Decl}_A}_{\mathit{Decl}_C}$ attribute is safely accessed.

Such an approach allows this framework to *gracefully degrade* in situations in which the generated backward transformation is partial, but the trees in the target can be manipulated (repaired) to be in the domain of the generated backward transformation. For real-world languages we expect there to be situations in which the techniques described in this paper cannot generate a total backward transformation, but the approach can still be used on those (hopefully large) portions of the language.

5. Related Work

Data transformations are an active research topic with multiple strategies applied on various fields, some with a particular emphasis on rule-based approaches. Czarnecki and Helsen in [4] present a survey of such techniques, but while they mention bidirectionality, they do not focus on it.

Bidirectional data transformations have been studied in different computing disciplines, such as updatable views in relational databases [1], programmable structure editors [9], model-driven development in software engineering [20], among others. In [5] a detailed discussion and extensive citations on bidirectional transformations are included.

A well-regarded approach to bidirectionalization systems is through lenses combinators [1, 7]. These define the semantic foundation and a core programming language, for bidirectional transformations on tree-structured data, but it only works well for surjective (information decreasing) transformations, whereas our system can cope with very heterogeneous source and target data types.

The approach followed in [14] uses a language for specifying transformations very similar to the one presented in this work, with automatic derivation of the backward transformation. Similar to our approach, this system statically checks whether changes in views are valid without performing the backward transformation, but they do not provided type-solving techniques such as provided by our *can-be*-based rule extension approach.

In the context of attribute grammars, Yellin's early work on bidirectional transformations in AGs defined attribute grammar inversion [27]. In attribute grammars inversion, an inverse attribute grammar computes an input merely from an output, but in our bidirectional definition of attribute grammars, a backward transformation can use links to the original source to perform better transformations. Thus, our approach can produce more realistic source trees after a change to the target.

6. Conclusion

In this paper we have shown how rewrite rules can be used to specify forward transformation, be automatically inverted to specify backward transformations, and then be implemented in attribute grammars where the quality of the transformation is enforced.

It is important to note that the features our bidirectionalization system supports are completely automatic for many application, freeing the programmer of having to write complex attribute equations that have to perform multiple pattern matching, manage both the links back and their types, prioritizing transformations, etc. The only part of our system which is not automatic are the tree repairs, but even in these cases we generate attributes to check for the need of repairs, further simplifying the programmers work.

We have implemented this approach in a Haskell prototype tool, available from `http://www.di.uminho.pt/~prmartins`. It has been used to implement all examples in this paper and automatically inverts the transformation and generates the attributes specifications for Silver [24].

There are few areas we plan to address. The first would be the generalization of this work to other attribute grammar systems beside Silver. We would like to generate AG specifications for other systems such Kiama [17], LRC [11] and JastAdd [6] or AG embeddings such as [13] so as to provide these techniques to a wider audience in the attribute grammar community.

That said, we would also like to implement these techniques as an extension to Silver, so that one can write these rewrite rules that define a forward transformation over Silver-specified grammars in the same Silver files that define the grammars. Silver is designed to be extensible so that new features, such as these rewrite rules, can be parsed as Silver specifications, analysed, and then translated down to core Silver specifications such as the attribute equations described above.

Finally, we would like to evaluate this approach on a number of mainstream syntactically rich languages. Once the Silver extension implementing these techniques is complete would use the approach on our Java [23], Promela [12] and ANSI C specifications.

Acknowledgements This work is partly funded by projects: ERDF through the programme COMPETE, project reference FCOMP-01-0124-FEDER-020532, by the North Portugal Regional Operational Programme (ON.2 ? O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), project reference RL3 SENSING NORTE-07-0124-FEDER-000058, by the Portuguese Government through FCT (Foundation for Science and Technology), and by the FLAD/NSF program Portugal-U.S. Research Networks 2011.

References

[1] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: A language for updatable views. In *Procs. of ACM Principles of Database Systems (PODS)*, pages 338–347. ACM, 2006.

[2] L. Chirica and D. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13(1):1–27, 1997.

[3] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I. *Theoretical Computer Science*, 17(2): 163 – 191, 1982.

[4] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[5] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Procs. of Theory and Practice of Model Transformations (ICMT)*, number 5563 in LNCS, pages 260–283. Springer-Verlag, 2009.

[6] T. Ekman and G. Hedin. The Jastadd extensible Java compiler. In *Procs. of ACM SIGPLAN Object-oriented Programming Systems and Applications (OOPSLA)*, pages 1–18. ACM, 2007.

[7] J. Foster, M. Greenwald, J. Moore, B. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

[8] G. Hedin. Reference Attributed Grammars. In *Proc. of Workshop on Attribute Grammars and their Applications (WAGA)*, pages 153–172. INRIA Rocquencourt, 1999.

[9] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Procs. of Partial Evaluation and Program Manipulation (PEPM)*, pages 178–189. ACM, 2004.

[10] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971).

[11] M. Kuiper and J. Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In *Procs. of Compiler Construction (CC)*, number 1383 in LNCS, pages 298–301. Springer-Verlag, 1998.

[12] Y. Mali and E. Van Wyk. Building extensible specifications and implementations of promela with AbleP. In *Proc. of Intl. SPIN Workshop on Model Checking of Software*, volume 6823 of *LNCS*, pages 108–125. Springer-Verlag, July 2011.

[13] P. Martins, J. P. Fernandes, and J. Saraiva. Zipper-based attribute grammars and their extensions. In *Procs. of Brazilian Conference on Programming Languages (SBLP)*, number 8129 in LNCS, pages 135–149. Springer-Verlag, 2013.

[14] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Procs. of ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 47–58. ACM, 2007.

[15] A. Middelkoop, A. Dijkstra, and S. D. Swierstra. Iterative type inference with attribute grammars. In *Procs. of Generative Programming and Component Engineering (GPCE)*, pages 43–52. ACM, 2010.

[16] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, 1989.

[17] A. M. Sloane, L. C. L. Kats, and E. Visser. A pure object-oriented embedding of attribute grammars. *Electronic Notes on Theoretical Computer Science*, 253(7):205–219, 2010.

[18] E. Söderberg. *Contributions to the Construction of Extensible Semantic Editors*. PhD thesis, Lund University, Sweden, 2012.

[19] E. Söderberg and G. Hedin. Circular higher-order attribute grammars. In *Procs. of Software Language Engineering (SLE)*, number 8225 in LNCS, pages 302–321. Springer-Verlag, 2013.

[20] P. Stevens. A landscape of bidirectional model transformations. In *Generative and Transformational Techniques in Software Engineering II*, number 5235 in LNCS, pages 408–424. Springer-Verlag, 2008.

[21] D. Swierstra, P. Azero, and J. Saraiva. Designing and Implementing Combinator Languages. In *Advanced Functional Programming*, number 1608 in LNCS, pages 150–206. Springer-Verlag, 1999.

[22] S. d. Swierstra and O. Chitil. Linear, bounded, functional pretty-printing. *Journal of Functional Programming*, 19(1):1–16, 2009.

[23] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *Proc. of European Conf. on Object Oriented Prog. (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer-Verlag, 2007.

[24] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, 2010.

[25] M. Viera, D. Swierstra, and A. Middelkoop. UUAG meets AspectAG: How to make attribute grammars first-class. In *Procs. of Language Descriptions, Tools, and Applications (LDTA)*, pages 6:1–6:8. ACM, 2012.

[26] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proc. of Programming Language Design and Implementation (PLDI)*, pages 131–145. ACM, 1989.

[27] D. M. Yellin. *Attribute Grammar Inversion and Source-to-source Translation*. Number 302 in LNCS. Springer-Verlag, 1988.