An Invasive Composition System for Local-to-Global Transformations

Thomas Cleenewerck [1,2]

PROG
Vrije Universiteit Brussel
Brussels, Belgium

Johan Brichau [3]

PROG
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

Transformation systems are particularly well suited to implement modular rules, transforming one language feature of the source language into a single or a composition of language features of the target language. However, in practice, transformation rules must be written which take one language feature and transform them into several language features belonging to various locations in the output program. The implementation of these so-called local-to-global transformations with rewrite rules is very complex and tightly coupled which imposes severe constraints on maintenance and evolvability. The four main coupling problems of the current-day implementations are presented and we indicate how these can be eliminated and reduced by our extension of the rewrite rule system. Furthermore we show how complex invasive compositions can be solved by abstract, reusable algorithms and mechanisms, rendering the implementation of local-to-global transformations into a semi-automatic process.

Key words: Maintainability, Evolvability, Rewrite rules, Program Transformations, Invasive Composition

[1] This research is partially performed in the context of the e-VRT Advanced Media project (funded by the Flemish Government) which consists of a joint collaboration between VRT, VUB, UG, and IMEC.
[2] Email: `tcleenew@vub.ac.be`
[3] Email: `jbrichau@vub.ac.be`

1 Introduction

The most dominating kind of transformations used today are rewrite rules [6,16,9]. They are very successful because of the simplicity of specifying a rule and the power of the underlying transformation system which already solves such complexities as scheduling [12] and provides adequate control structures such as pattern matching [15]. A rewrite rule specifies a substitution of a part of the source abstract syntax tree (AST) by a part of the target AST. Consequently, rewrite rules are modular with respect to the grammar of both the target and source language and with respect to other rules. This means that a rewrite rule is only concerned with transforming a part of the tree, without context information. It can consequently be reused in other implementations where the context of a source AST might differ. However, implementations consisting merely of such 'in place' transformations are hardly ever expressible. Quite often, a single transformation must exert some output on other parts of the target AST. In the survey of Jonne van Wijngaarden et. al. [15] these kinds of transformations are called *local-to-global* transformations.

The implementation of local-to-global transformations solely by means of rewrite rules is very complex and tangled. Extensions of transformation systems to the rewrite rule paradigm with various data acquisition techniques like queries and tree traversals [14] did not reduce the tangling. We found that, similar to other software development techniques, this tangling severely jeopardizes future maintenance and evolution.

In earlier work [4] we proposed an extension to rewrite rules system which disentangles the implementation. This was achieved by providing a semi-automatic process that treats the outputs which cannot be handled by the currently available 'in-place' substitution mechanism. In this paper, we identified the 4 main coupling problems of the current-day implementations and show how these can be amended by our approach. Furthermore we show how complex invasive compositions can be solved by abstract and reusable integration strategies.

In the following section, we investigate the fundamental problem that forces a developer to write tangled rewrite rules. Section 3 introduces our solution architecture, which explains the architecture in more detail. In section 4 the architecture is evaluated and the implications of the approach are discussed. The related work is discussed in section 5 and the paper concludes in section 6.

2 Local-to-Global Transformations

We conceived a small domain specific language (DSL) to illustrate the problems with the implementation of local-to-global transformation with rewrite rules. Afterwards we generalize these specific examples and distill the fundamental causes which cripple maintainability and evolvability.

```
W  wizard( (
P    page (
I      inputfield("appname" defvalue("T"))

I      inputfield("appdir"
D        defvalue( stats(( return( +(+(
K                (key(programpath) "/") appname)) )))))
   )))
```

Fig. 1. The source AST of installation program written in WIZ.

2.1 Motivating Example

The small DSL we use as an example is called the WIZ language. It was designed to describe installation wizards. A program written in WIZ is transformed into a Java program on top of a framework. The purpose of WIZ is to hide the developers from tedious glue code and to encapsulate the best practices to instantiate the underlying framework. The surface syntax of WIZ is of no importance for this paper, instead we describe how one uses the language and how the transformations on the AST implement the translation to Java.

The example WIZ program of figure 1 describes a wizard consisting of one page. The page contains an inputfield for the application name (`appname`) and an inputfield for the installation directory (`appdir`). The default value of the installation directory is the result of the concatenation of the registry key `programpath` (holding the default location for applications) and the application name (`appname`). When the application name is changed, the default value for the programpath (which depends on it) is recomputed. The desired result after transforming this specification is shown in figure 2A.

The implementation of the WIZ compiler is divided in 5 transformations. Each of the transformations corresponds to the translation of one source AST node to its Java counter parts. The source AST nodes are `wizard`, `page`, `inputfield`, `defvalue` and `key` which are respectively translated by the W, P, I, D, K transformations. In figure 1 the nodes are prefixed by the transformation that takes the prefixed node as input. The result nodes produced by those transformations are shown in figure 2A and are also prefixed using the same prefix. The figure 2A clearly shows that the target AST nodes, produced by the transformations, are scattered in the desired target program. This scattering is not an accidental situation, but rather a structural problem between the two languages. The resulting nodes of one transformation may not all substitute the source node. Only results of a particular type may be substituted with the source node, to ensure the resulting AST is type correct. The results that do not fit must be put in other places in the tree. For example, the source `key` node is part of an expression, therefore the results the `key` transformation produces must be an expression in order to yield a correct target program. Its other results, the datamember and the statement, must consequently be put in other locations of the target program. Besides typing problems, some result nodes belong to a different part in the target program due to grammatical constraints or to particular semantical constraints

```
1  W class (datamember("Wizard" "wizard")
2  I  datamember("Inputfield" "appname")
3  I  datamember("Inputfield" "appdir")
4  K  datamember("Key" "programpath"))
5  W  method("main" (...) (stats (
6  K    assign( var("programpath") new("Key"))

7  W    assign( var("wizard") new("Wizard"))
8  P    assign( var("page") new("Page"))

9  I    assign( var("appname") new("InputField"))
10 I    assign( var("appdir") new("InputField"))
11 I    mcall( var("appname") setListener
12 I        (... mcall( () "appdirdefvalue" ()) ... ))
13 I    mcall( var("appname") setValue ("T"))
14 I    mcall( var("appdir") setValue
15 D         (mcall( () "defvalue" ())))

16 P    mcall( var("page") "add" (
17 I         var("appname")))
18 P    mcall( var("page") "add" (
19 I          var("appdir")))

20 W    mcall( var("wizard") "add" (
21 P         var("page")))
22 W   )))
23 D   method("defvalue" () (stats (
24 D      return( +(+( mcall(
25 K              var("programpath") "getValue" ()) "/")
26 D          mcall( var("appname") "getValue" ())))
27 D   )))
```

```
1 class (datamember("Wizard" "wizard")
2  datamember("Inputfield" "appname")
3  method("main" (...) (stats (

4   assign( var("wizard") new("Wizard"))
5   assign( var("page") new("Page"))

6    assign( var("appname") new("InputField"))

7    mcall( var("appname") setValue ("T"))

8    mcall( var("page") "add" (var("appname")))
9    mcall( var("page") "add" (
10     CONS( LOCAL( var("appdir")
11     (NONLOCAL(
12       assign( var("appdir") new("InputField"))
13       mcall( var("appname") setListener
14           (... mcall( () "appdirdefvalue" ()) ... ))
15       mcall( var("appdir") setValue
16         CONS( LOCAL( mcall( () "defvalue" ()) )
17         (NONLOCAL(
18            method("defvalue" () (stats (
19            return( +(+( CONS(
20             (LOCAL( mcall( var("programpath") "getValue" ()) "/"))
21             (NONLOCAL( datamember("Key" "programpath")) )
22              NONLOCAL( assign( var("programpath") new("Key")) )))
23             mcall( var("appname") "getValue" ())))
24           )))
25         ))
26       )))
27      NONLOCAL( datamember("Inputfield" "appdir") ))))))

28    mcall( var("wizard") "add" (var("page")))
29  ))))
```

Fig. 2. (A) The left part of the figure shows the target AST obtained after transforming the sample installation program (figure 1). (B) The right part shows the state of the target AST right before the relocation and integration of the nonlocals produced by the inputfield transformation is initiated.

imposed by the DSL developer.

2.2 Implementation using rewrite rules

Rewrite rules describe 'in-place' substitutions, the results produced by the rewrite rule must all fit into the location of the node under transformation, which we will refer to as the *local results*. In order to deal with the other results that do not fit in the location, the so called *non-local results*, we need a workaround. One can think of several workarounds but they all boil down to the following strategy. Because most of the rewrite rule systems have been extended with some kind of traversal mechanism [14], we will only consider such systems in the discussion of the workaround. A local-to-global transformation is broken up into several rewrite rules and traversals, each one responsible for a single result. The implementation of each transformation takes three steps. First, a primary rewrite rule must be written that produces both local and non-local results. The results are contained in a tuple (`CONS(... , ...)` cells. Second, for each non-local result write a secondary traversal that collects the non-local. Third, the collected non-local must be fed into a secondary rewrite rule that matches the correct target location where the non-local must be integrated.

Although the primary rewrite rules are modular, they construct an intermediate representation (IR) which is an new structures that must be additionally maintained whenever the language evolves. Consequently this solution only increases the maintenance efforts. Experiments with such an approach showed that the transformation from Cobol to Java required about 70 IRs [CDMS02], introducing enormous maintenance efforts. Moreover, the problem has merely been shifted to the transformations that create the intermediate structure, and therefore no real solution has been created.

2.3 Coupling and tangling

In various software development methodologies one of the most common approaches to facilitate evolution and maintenance is by minimizing the tangling and coupling within an implementation. By doing so, the components of an implementation can be more easily replaced by another one and can be reused in other implementations. Since the implementation of a set of transformations can be regarded as a normal software artifact, the reduction of the tangling and coupling hold the same premises. In the reminder of this section the coupling and tangling issues of the workaround strategy are discussed.

The source of the tangling and coupling problems of this strategy are the dependencies of the secondary traversals and rewrite rules with the rest of the rewrite rules. We distinguish the following 4 coupling problems:

(i) The mechanisms of the workaround are interwoven with the concrete implementation of the local-to-global transformations to the extent that they are hard to distinguish within their implementation. Furthermore, their implementations are manual and are degraded to a more functional (stating rather how than what) approach.

(ii) The coupling of the secondary traversals and rewrite rules and the outcome of the primary rewrite rules.

(iii) The scheduling problems of the secondary traversals and rewrite rules in order not to violate the context conditions of other secondary traversals and rewrite rules.

(iv) The tight coupling of the secondary traversals and rewrite rules on the exact properties of the AST at a certain stage during transformation.

The underlying problem causing these 3 later dependencies is that the secondary rewrite rules and traversals all operate on a tree that has been fully rewritten into target nodes. As a consequence, their matching clauses are entirely expressed in terms of target nodes. Therefore any small change in the grammar of the source language or in the implementation of the transformations due to evolution or maintenance easily breaks most of the dependencies made in the implementation of the secondary traversals and rewrite rules. Let us further discuss each of those dependencies in more detail using the figure 2B .

The first problem is *the tight coupling of the secondary traversals and* *rewrite rules and the outcome of the primary rewrite rules.* Consider the secondary traversals and rewrite rules to collect and move the non-locals of the inputfield transformation: the two non-local statements (an assignment (line 12) and two method calls (lines 13 and 15)) of the topmost cons cell (line 10). These two nonlocals must be moved to the enclosing `method` node. Writing such a traversal heavily depends on the context node (line 9) produced by the page transformation and the non-locals of the inputfield transformation (line 12-15), in order to make sure that the correct non-locals are matched. Like the traversals, the left hand side, the triggering condition and the right hand of the secondary rewrite rule that integrates the nonlocal on the correct location, is also expressed in terms of target nodes, which results in the same deficiencies.

The second problem is *the scheduling problems of the secondary traversals and rewrite rules in order not to violate the context conditions of other secondary traversals and rewrite rules.* Let us take a look at the scheduling of the secondary traversal and rewrite rule of the inputfield transformation and those of the default value transformation and the key transformation. The latter handle the non-locals in the nested cons cells of the cons cell (line 10, produced by the inputfield transformation). The correct location of the non-local statement (line 22) produced by the `key` transformation is in the beginning of the `main` method. However, depending on the strategy to deal with the cons cells the non-locals might end up in different locations. A breadth-first (top-down) strategy, places the non-local statement (line 22) inside its enclosing method `defvalue`. But a depth-first (bottom-up) strategy to deal with cons cells, places the non-local statement (line 22) inside the `main` method. The implementation of these secondary traversals and rewrite rules thus depend on the chosen scheduling and vice-versa.

The third problem is *the tight coupling of the secondary traversals and rewrite rules on the exact properties of the AST at a certain stage during transformation.* Since the secondary traversals and rewrite rules continuously change the tree, their specifications depend on a particular stage during transformation. Consider for example the integration of the different non-local statements produced by the inputfield transformation: assigning and creating the new field (line 12), attaching listeners to other dependent inputfields (lines 13) and initializing the inputfield with a default value (line 15). Although those statements are in the right order from one single inputfield viewpoint, when there are two or more inputfields these statements need to be regrouped. It is obvious that first *all* the inputfields must be created before initializing or configuring them, that secondly *all* the listeners must be attached to each inputfield so that thirdly when the inputfields are initialized *all* dependent inputfields get properly initialized. The implementation of the regrouping is not simple and heavily relies on the contents of the `main` method. The statements of the `main` method body are almost exactly matched by the rewrite rule in order to determine the correct position where to integrate the creation of the new inputfields, the attachement of the listeners and the initiallization of the inputfields. Moreover, since the contents of the `main` method is constantly changing by moving and integrating non-local statements, the integration rewrite rule only works on a particular point the transformation process.

3 Resolution of Non-local Target Nodes

Our solution architecture modularizes the three responsibilities of local-to-global transformations in 3 steps: (1) a generation step that produces both local and non-local target nodes, (2) a relocation step which navigates through the target AST to find the location of the non-local nodes, (3) an integration step to add the produced target AST nodes in that location.

The last two responsibilities are handled by a composition system that is driven by a declarative specification stating the target location of each non-local and the integration of the non-locals into the tree. Some parts of the required declarative specifications can be automatically derived from the target language grammar, hence yielding a semi-automatic composition system. In the following subsections, we describe each of the steps in detail. As the architecture is considered an extension of existing systems, the large amount of existing systems and their various natures rapidely complicates a discussion of its implementation. Therefore, a full discussion of the implementation of this architecture falls outside the scope of this paper. However each subsection clearly states the mechanisms that are required to implement the architecture.

3.1 Generation Step

To distinguish the non-local target nodes from the local target nodes in the rewrite rule, the non-local nodes are attached to a special `NONLOCALS(...)` node, which is in turn, together with the local target nodes, part of a cons cell. The example code below shows how the `key` transformation can be written as a normal rewrite rule.

```
KEY(NAME) --> CONS(LOCAL( mcall( var(NAME) "getValue" ()) )
        NONLOCAL(( "datamember", datamember( "Key" NAME) )
                 ( "initialize", assign( var(NAME) new("Key")) ) )
```

A crucial part of the system and difference between the workaround is that non-locals can carry any kind of information. This is heavily exploited by the semi-automatic composition system to identify the non-locals and to guide the integration process of the non-local (see section 3.3.2). To ease and standardize the identification of the nonlocal, nonlocals always have at least a name or an identifier. In contrast to the common approaches dealing with names and identifiers, these need not to be unique within a rewrite rule nor amongst other rewrite rules. The advantage of this naming schema is that the name or identifiers can be used as a grouping mechanism, abstracting the actual number of nonlocals. The additional information can be used to select subgroups adhering to certain criteria. In the above example, the name of the first nonlocal is `datamember`, and the second nonlocal is `initialize`. No other additional information is provided, since their names suffice to distinguish between both of them.

3.2 Relocation Step

The relocation step is executed after the execution of all modular rewrite rules. In this step, the non-local target AST nodes are moved to their correct location in the target AST. The correct location for the non-local AST nodes is determined by (1) the grammar of the target language and (2) custom semantics which is specific for the desired target program. The relocation for non-locals, which correct location is based the grammar of the target language, is now driven by the target language grammar and can even be automated. The relocation for non-locals, which correct location is based on custom semantics, needs to be written manually.

3.2.1 Automatic Relocation

In general, a non-local `NL` can be integrated in a node `E` if `NL` can be a child of `E` according to the grammar of the target language. Each non-local target node is lifted upwards in the AST until a node `E` is found where the nonlocal `NL` can be integrated in `E'` or in one of the indirect children of `E'`. The correct child is determined by a depth-first, left-to-right strategy. More details of the algorithm can be found in [4].

Let us illustrate how this automatic relocation process works for the inputfield transformation. The state of the parse tree at that time is shown in figure 2B. The inputfield transformation produced a couple of non-local statements and a non-local datamember. The automatic relocation process for the datamember will move the datamember upward in the tree until it finds a node which, according to the grammar of Java, can contain other datamembers. The first node which is capable of doing so, is the `class` node (line 1). Similarly, the automatic relocation process for the statements will yield, the `stats` node (line 3) (i.e. a node that holds several statements) of the `main` method node (line 3). The actual integration of the non-local nodes will be performed by the integration step explained in the following section.

Automatic relocation only works in cases where the location of the non-locals can be determined unambiguously or in cases when there is a simple satisfying strategy to disambiguate. In case of domain specific languages (DSLs) automatic relocation is very well suited. DSLs bear a lot of language constructs that may only be used in a certain context, so disambiguation is not often needed. The technique is also useful for target languages which are general purpose languages. Classes, datamembers, methods and statements are typical language constructs that are only allowed in a specific context. If the location of other language constructs like expressions and statements cannot be unambiguously determined but the nearest possible location is often the desired one. However, automatic resolution fails for most types of non-locals in languages, like for example HTML, where most language constructs may be used in every other language construct. For these languages custom relocation must be used.

3.2.2 Custom Relocation

In case automatic relocation fails or the relocation is dependent on the particular semantics of the target program, the custom relocation must be defined by the user. A custom relocation path for a non-local is not specified at the time when the non-local is produced. Instead the path is attached to the non-local afterwards, by means of a 'special' rewrite rule of the following form:

```
NONLOCAL( _name of the nonlocal_ , NODE) OF _source ast node_
     --> NONLOCAL( _name of the nonlocal_ , RELOCATE(NODE, _relocation path_))
```

The parts enclosed in underscores are merely place holders for the values they denote. The non-local `NODE` produced by a certain source AST node is referred to by its name and the source AST node. It is rewritten into a non-local with a custom relocation path.

The relocation paths are also expressed in terms of the source AST. Consequently, the relocation paths express that the non-local node should be relocated to the top level target AST node, which is the result of transforming the source AST node pointed to by the custom relocation path. The further integration into the subtree of the top level target AST node is accomplished by means of integration rules (next section).

Let us illustrate the above with an example. Consider the non-local statement (line 22 in figure 2B) produced by the `key` transformation which initializes the non-local datamember. Since the datamember must only be initialized once, the proper place of the initialization statement is at the beginning of the `main` method.

```
NONLOCAL(INITIALIZE, NODE) OF KEY --> NONLOCAL( INITIALIZE, RELOCATE(NODE, //wizard))
```

The above rule will substitute the initialize non-local node (denoted by `NONLOCAL(INITIALIZE, NODE)`) produced by transforming the `key` source AST node, into a non-local with the specification path attached to it. The path `//wizard` points to the `main` method target AST node obtained by transforming the source AST ancestor `wizard`. The language to express the path is XPath [1].

3.2.3 Discussion

Instead of specifying the relocation in terms of the target nodes, we specifying the relocation with 'special' rewrite rules in terms of the source AST node and the name of the nonlocal, which attaches the nonlocals after they are produced. This decoupling between the relocation path and the non-local prohibits a strong dependency between the rule that produces the non-local and the rest of the language implementation. Hence, the dependencies, that resulted from the second coupling (section 2.3), between the rewrite rules that produce the non-locals with the other rewrite rules are eliminated. By eliminating these dependencies, the changes to the rest of the language implementation (grammar and rewrite rules) will no longer invalidate the rewrite rule producing the non-local but will only affect where the non-local result must be relocated to, hence will only have an affect on the relocation path. Naturally, in order to attach the relocation of a nonlocal, we need to establish some kind of dependency. The 'special' rewrite rule establishes only a dependency on the AST source node and the name of the nonlocal. This is a weaker dependency which only depends on the AST source node and on the fact that a transformation of the source AST node produces a nonlocal identifiable by a name. As a result the rule is more maintainable and more robust to future evolutions of the language implementation i.e. changes to other rewrite rules, results they produce, etc.

Paths on top of the source AST are more robust to changes to the produced target AST caused by modifications to the rewrite rules which produce the target nodes, by the relocation phase and by the subsequent integration phase (which is described next). Moreover there is no detailed knowledge required to further integrate the non-locals into the subtree of the produced top level target node AST node. The actual integration is captured in separate integration rules. This minimizes the dependencies between the rewrite rule that relocates the non-local node and the rewrite rule that produces the top level of AST node.

3.3 Integration Step

In the previous section we saw that the relocation moves the non-locals to the top-level target AST nodes produced by the transformation of the source AST node described in the relocation. In this final step the non-locals are invasively integrated into that subtree (starting at the top-level target AST node).

3.3.1 Approach

Integration ensures that the non-locals are correctly integrated in the subtree pointed to by the relocation path according to the semantics of a particular integration and to particular language constraints. For example, the integration of the non-local statements, produced by the inputfield transformation, into the `main` method body must obey the user-defined order of the different statements (first creation of *all* inputfields, followed by the *attachment* of listeners and finally their initialization with default values). Another example is the integration of the non-local datamember, produced by the same transformation, into the `main` class, which must enforce the uniqueness constraint of the datamembers within a class.

Integration is a close interplay of three different policies: integration, correspondence and combination. The integration policy is called for each non-local that was relocated to an AST node. Its task is to integrate the non-local in the AST node itself or in one of its children. The correspondence policy is used by the integration policy to check if nodes in the AST need to be combined with the non-local. The correspondence policy is more like a predicate defined on nodes to check if they correspond. The combination policy performs the combination between nodes.

An integration for a nonlocal is specified by a triple (see example below) of an `integration`, `correspondence` and `combination` node containing the name of the rule and if necessary some additional information.

```
(INTEGRATION(POLICY1), CORRESPONDENCE(POLICY2), COMBINATION(POLICY3))
```

Similar to the custom relocation paths, the specific integration rules to be used are not specified at the time when the non-local is produced. Instead the particular rules to be used are also attached to the non-local afterwards by means of the 'special' rewrite rules, rendering thus the same evolution and maintainability benefits. The rules to be used are attached to the non-local by means of a 'special' rewrite rule of the following form:

```
NONLOCAL( _name of the nonlocal_ , NODE) OF _source ast node_
    --> NONLOCAL( _name of the nonlocal_ ,
                  ((INTEGRATION(POLICY1),
                  CORRESPONDENCE(POLICY2),
                  COMBINATION(POLICY3)) ,NODE))
```

3.3.2 Integration

We will start with the most simple case of an integration rule which are the generic integration rules. Afterwards, the custom integration rules are explained by a more complicated example.

Although the integration depends on the particular non-local and the particular target subtree, our architecture provides a generic integration rule that is instantiated for each kind of target AST node. Generic integration rules alleviate the developer of tedious integration rules for example to traverse intermediate structures. The generic integration strategy cannot be written directly in rewrite rules and needs to be generated for every kind of target AST node. The strategy for the an AST node `NODE( NODE1*, NODE2*, ..., NODEn* )` first determines in which part `NODEi*` the non-local belongs starting from the left most part. Second, a node `NODEij` of the `NODEi1, ..., NODEim` corresponding with the non-local node is searched. When a corresponding node has been found, the node and the non-local is combined. Otherwise the non-local is appended to the part `NODEi*`.

The instantiation of the generic integration strategy for a `class` node is shown below. The two rules integrate a non-local `NODE` with the default integration strategy (line 6) into the child nodes `NODE*` of the `class` node. When one of the `NODE*` nodes corresponds using `POLICY2`, the non-local node `NODE` is combined with it (line 8), otherwise the non-local is just added to the body of the class (second rule, line 11).

```
(1)   NODE* = NODE1* NODE2  NODE2*
(2)   CORRESPOND(POLICY2, NODE2 , NODE) = TRUE
(3)   ===============>
(4)   INTEGRATE(DEFAULT,CLASS(NODE*),
(5)      NONLOCAL( NAME ,
(6)                ((INTEGRATION(DEFAULT),
(7)                  CORRESPONDENCE(POLICY2), COMBINATION(POLICY3)) ,NODE))
(8)       =  CLASS(NODE1* COMBINE(POLICY3, NODE2, NODE) NODE2*)
(9)   INTEGRATE(DEFAULT,CLASS(NODE*),  NONLOCAL( NAME ,
(10)             ((INTEGRATION(DEFAULT), CORRESPONDENCE(POLICY2),
(11)               COMBINATION(POLICY3)) ,NODE))) =  CLASS(NODE* NODE)
```

Note that the generic integration rules are made possible because the integration is divided into a integration rule, a combination rule and a correspondence rule which all three are implemented by a different set of rewrite rules.

Naturally, custom integration rules can be provided as well. The integration for the non-local statements produced by the inputfield transformation must regroup the statements so that firstly all the inputfields are created, secondly the listeners are installed and finally the inputfields are initialized with a default value. We will exploit the openness of our system, in particular the fact that each non-local can carry an arbitrary amount of information to guide the integration process (see 3.1). By attaching to each non-local statement an identifier, we can easily group during integration the statements. The order of the groups can be specified by additional information which can be interpreted by the integration rule.

Below the two rewrite rules that implement this integration rule are shown. The non-local node `NODE` is annotated with the group (`NAME`) to which it belongs, and a triplet specifying its integration. The integration contains besides the name of the integration rule (`GROUP`) also the name of the group of statements (`NAME2`) that must come before it. The first rule inserts the non-local `NODE` into the list of statements (`STATS`) before the first statement belonging to the same group (`NAME`) as the non-local does. When there is no such statement, the second rule inserts the non-local right after the last statement of the group of statements that must come before it.

```
(1)  INTEGRATE( GROUP,
        STATS( CONS1*  CONS(NODE1,NAME) CONS2* ),
        NONLOCAL( NAME ,
               ((INTEGRATION(GROUP, NAME2),
                 CORRESPONDENCE(DEFAULT), COMBINATION(DEFAULT)) ,NODE)))
        =  STATS(CONS1* CONS(NODE,NAME) CONS(NODE1,NAME) CONS2*)

(1)  INTEGRATE( GROUP,
        STATS( CONS1* CONS(NODE1,NAME) CONS2* ),
        NONLOCAL( NAME ,
               ((INTEGRATION(GROUP, NAME2),
                 CORRESPONDENCE(DEFAULT), COMBINATION(DEFAULT)) ,NODE)))
        =  STATS(CONS1* CONS(NODE1,NAME) CONS(NODE,NAME2) CONS2*)
```

The implementation of the regrouping of the non-local statements by the above set of integration rules is not a particular integration for a particular transformation. The integration rules are abstracted from the particular regrouping required for the statements because they only implement a partial ordering. The actual ordering is driven by the declarative information attached to the non-locals. So, the integration rule can be reused for similar cases where partial ordering is needed.

3.3.3 Correspondence

The integration of non-locals is bound by target language constraints and custom semantics. For example, a datamember can't just simply be added to the main class, because a Java class may not contain two datamembers sharing the same name. So during the integration (see section 3.3.2) we needed to compare the non-local datamember with the other datamembers to ensure this constraint. Correspondence rules are declarative specifications stating whether two nodes correspond. They are implemented as rules which transform a `CORRESPOND` node, containing a name and the two nodes that need to be compared, into a `TRUE` node when the two nodes do correspond. The correspondence rule below, called `DEFAULT` states that two datamembers correspond when both their `name`'s are the same.

```
CORRESPOND(DEFAULT, DATAMEMBER(NAME,TYPE), DATAMEMBER(NAME,TYPE2)) = TRUE
```

As you can see, correspondence rules are totally independent of the implementation of the WIZ language and only involve target language nodes.

3.3.4 Combination

Combination rules are declarative specifications that combine two AST nodes together in one node. They are implemented with rewrite rules that operate on a `COMBINATION` node. Like the `CORRESPOND` node, a `COMBINATION` node contains three parts: a name, and the two nodes that need to be combined. The first node is the non-local node and the second node is the node that is part of the AST. The result of a combination is a new node.

We cannot illustrate the combination rules using the current implementation of the WIZ language because there are no combination rules needed. However, if we use the an alternative implementation for the `key` transformation, a combination rule for methods is needed.

```
KEY(NAME) --> CONS(LOCAL( mcall( var(NAME) "getValue" ()) )
        NONLOCAL(( "datamember", datamember( "Key" NAME) )
                 ( "initialize", method("main" () (stats ( assign( var(NAME) new("Key"))))) ) ))
```

Instead of producing a non-local assignment statement to initialize the nonlocal datamember, the rewrite rule produces a nonlocal `main` method. The relocation process lifts the `main` method node to the root node class, where subsequently the `main` method node is integrated by the integration process. Because a Java class cannot contain two methods with the same signature, a correspondence between the two `main` method is found. Consequently, the bodies of the two methods must be combined. There are of course various ways of combining two method bodies. Each of them can be encoded in a separate rewrite rule. In the `ONTOP` combination rule below for the two `main` methods, the non-local method body is placed before the other method body.

```
COMBINATION(ONTOP,
            METHOD(NAME, ARGS, STATS(STATEMENT*)),
            METHOD(NAME, ARGS, STATS(STATEMENT*2)))
            = METHOD(NAME, ARGS, STATS(STATEMENT* STATEMENT*2))
```

The above rule simply concatenates the method bodies without taking possible variable declaration clashes into account, or differences in the parameter names of the methods. The later can be amended by proper reconciliation of the deviating method body. The former can be amended by extending the rule with a simple traversal and a condition (rule shown below). To avoid duplicate variable declarations within the scope of the merged methods, the intersection of the variable declarations of the two methods must be empty (line 3). Calculating the set of variable declarations is done by a top-down traversal of the bodies of the two methods where each variable declaration is collected a set (line 1). The traversal breaks when other nodes are encountered (line 2) to prohibit traversing into inner-scopes.

```
(1) VARDECLS(VARDECL(VAR,TYPE,EXP),VARDECL*) = VARDECL* VARDECL(VAR,TYPE,EXP)
(2) VARDECLS(NODE,VARDECL*) = VARDECL*

(3) intersection( VARDECLS(STATS(STATEMENT*, []),
(4)                       VARDECLS(STATS(STATEMENT*2, [])) ) = []
(5) =========================================================
(6) COMBINATION(ONTOP,
            METHOD(NAME, ARGS, STATS(STATEMENT*)),
            METHOD(NAME, ARGS, STATS(STATEMENT*2)))
            = METHOD(NAME, ARGS, STATS(STATEMENT* STATEMENT*2))
```

Like the correspondence rules, the combination rules are also stated in separated rules and involve nothing more then the combination of two nodes. The advantages those combination rules are thus also the same as the advantages of the correspondence rules: modularity and reusability resulting in a reduced implementation effort and a positive impact on the maintenance and evolution of compiler implementations.

3.3.5 Discussion

The correspondence check facilitates many necessary and interesting mechanisms. The two most important mechanisms are the combination of partial results and the enforcement of uniqueness constraints. By establishing a correspondence between the two partial `main` methods, these got combined together into a complete method during the integration. By establishing a correspondence among datamembers nodes (section 3.3.3), we were able to enforce that datamembers within a class must be unique.

The separation of the integration process yields integration, correspondence and combination rules which are nicely separated into different concerns. The advantages of this separation are twofold: increased reusability and maintainability. The following three arguments support this claim.
First, integration can be more easily customized and tailored by providing, for each of the concerns, an arbitrary number of rules. These alternatives

work seamlessly together with the other already existing rules, enabling the developer to reuse much of the already existing integration logic. One example illustrating the advantage of this tailorization are the generic integration rules. Because integration is captured in a separate concern, we were able to provide automatically generated integration rules that could be configured with custom specified correspondence and combination rules.
Second, the correspondence and combination rules only involve target language nodes, rendering them reusable even across several implementations which share the same target language. Even the integration rules can be abstracted to the level of general integration strategies by driving them with declarative information which is attached to the non-locals.
Third, the integration process of the non-locals does not depend on a certain stage during the integration process. This is achieved by two properties: incrementability and declarativeness. First, the integration rules incrementally integrate non-locals in the produced target AST. This way we can relax the requirements of the integration on a particular state of the target AST. Second, the integration rules are driven by information which is attached to the non-locals. This allowed us to abstract the integration rule from the particular integration problem at hand to a more general rule. Hence, these two properties reduce the dependencies, that resulted from the fourth coupling (section 2.3), of an integration on the particular state and properties of the AST in the transformation process.

3.4 Putting it all together

What follows is a summary of the steps involved and the development effort required to implement local-to-global transformations. The inputfield transformation is used as an example.

In the first step a rewrite rule transforms the inputfields (shown below) into five target AST nodes: one local node and four nonlocals. The three nonlocal statements are wrapped by a triplet containing their names, the integration nodes and the nonlocal node itself. The order of the statements is specified by the integration called `GROUP` and their partial order can be specified by their group name (e.g. `"link"` for the second nonlocal) and the group name that must precede it (e.g. `"create"` for the second nonlocal). Since ordering is can be specified based on the implementation of the inputfield transformations only, the integration, correspondence and combination can already be specified during generation. Hence there is no need to attach the integration, correspondence and combination with the special rewrite rule 3.3. How the code that produces the links (in the form of listeners) with the other inputfields is generated, is a technical detail. We only included it for a full understanding of the rewrite rule. The `BUILDLISTENERS` is a rewrite rule that traverses the `DEFVALUE` source node (first argument) and collects the statements in its last argument `STATEMENTS*`. The second argument `FIELDNAME` denotes the name of the variable to attach the listeners to and the third argument (`DEFVALUE`) will yield after transformation the appropriate code to reinitialize the variable.

```
INPUTFIELD(FIELDNAME DEFVALUE) =
     CONS( LOCAL( var( FIELDNAME ))
         NONLOCAL( "create" ,
                ((INTEGRATION(GROUP, ""),
                  CORRESPONDENCE(DEFAULT), COMBINATION(DEFAULT)) ,
                  assign( var("appdir") new("InputField"))))
         NONLOCAL( "link" ,
                ((INTEGRATION(GROUP, "create"),
                  CORRESPONDENCE(DEFAULT), COMBINATION(DEFAULT)) ,
                  BUILDLISTENERS(DEFVALUE, FIELDNAME, DEFVALUE, []) ))
         NONLOCAL( "initialize" ,
                ((INTEGRATION(GROUP, "initialize"),
                  CORRESPONDENCE(DEFAULT), COMBINATION(DEFAULT)) ,
                  mcall( var("appdir") setValue DEFVALUE )))
       NONLOCAL( "datamember", datamember("Inputfield" "appdir") ))
BUILDLISTENERS(VAR, FIELDNAME, DEFVALUE, STATEMENTS* )  =
        STATEMENTS*  mcall( var(VAR) setListener (... DEFVALUE ... ))
```

In the second step, the automatic relocation of the statements and datamembers yields their correct position in the target tree. There is thus no need to write a custom relation path.

In the third step the non-locals of the `inputfield` transformation must be integrated into the AST nodes designated by the relocation step. For statement and datamember node types we define a correspondence rule called default stating that statements never correspond and datamembers correspond when their names and types are equal (see 3.3.3).

```
CORRESPOND(DEFAULT, STATEMENT1, STATEMENT2) = FALSE
```

Subsequently the integration rules must be defined to integrate the statements, the datamembers and the methods. As the architecture provides a generic strategy for integration rules for each target AST node, consequently only integration rules that deviate from the generic strategy need to be written. For the inputfield transformation only an integrate rule for regrouping the statements had to be written (see section 3.3.2). The integration of the datamember is handled by the generic strategy.

4 Discussion

The process of handling non-locals is divided into a number of separated modules: the generation, the automatic relocation, the custom relocation, and the three integration, combination and correspondence rules. Each part addresses a tangible and separated concern of the implementation and captures its intention in a clear declarative way. We can thus conclude that the first coupling (section 2.3) between the mechanisms to implement local-to-global transformations and a particular implementation for a transformation, has been eliminated. The other couplings mentioned in the beginning of the paper have been adressed and discussed in the previous sections 3.2.3, 3.3.5.

The decoupling and disentangling of the implementation relies on the correct use of names and the extra information attached to the nonlocals. The names themselves do not contain any semantics. Hence it is up to the user to properly use the names to guide himself in the implementation of the local-to-global transformations. A specification language to express these semantics would be more solid.

By reducing and eliminating the coupling in the implementation maintenance and evolution is facilitated. The impact of maintenance and future evolutions is limited to specific modules. Changes to the source language have only an impact on the custom relocation. Changes to the target language have only an impact on the three rules (integration, correspondence and combination rules) and on the automatic relocation. Since the latter is an automated process based on the target language, the changes to the target language are automatically reflected in the automatic relocation. Finally changes to the transformations of the source language to the target language are localized to and affects only the implementation parts of that transformation i.e. generation, relocation and integration. During generation the rewrite rule may produce all the results (both local and non-local) that are derivable from the source node under transformation without taking other transformations into account. If custom relocation was necessary, attaching the relocation path and the path itself is only dependent on the fact that one transformation produces a result with a particular name, instead of all the transformations. The integration rules are solely defined on top of the target language nodes which yield them also independent of the other transformations.

However, other factors like complexity and comprehensibility also influence maintainability and evolvability. Although the architecture is more complex, it absorbed procedural aspects of the implementation leaving a greater concentration of declarative aspects in the implementation. But that does not necessarily render implementations more comprehendible to the programmer. The impact of this on maintainability and evolvability is difficult to deduct. A case-study of several complete transformations sets comparing the different contemporary approaches is required to clarify this issue.

There are a couple of limitations of our approach. First, the automatic relocation process is limited to the grammar of the target language. Using more information about the semantics of the target programming language could improve the system. Second, the integration process is incremental. This incrementability increases the complexity of the integration rules. A full discussion of these issues can be found in [4].

The validation of our approach is based on early experiments including the implementation of the compiler for the WIZ language used in this paper. We are currently involved in several experiments to validate our technique: the development of a DSL for quizzes with the flemish broadcasting company [5], the development of a open set of language components for a family of DSLs to describe advanced transaction models [7], and the development of modular language features for developing EJB specifications. To implement these languages local-to-global transformations and other invasive composition mechanisms are needed. These languages will be used as a case study and serve as an empirical validation for our approach. The mechanisms proposed in this paper were implemented on top of the linglet transformation system (LTS) (previously known as the keyword based programming toolkit [3]). The modularity of the system immediately acted as additional validation mechanism, testing whether the implementation of our semi-automatic mechanism obeyed this. Since modularity reduces the dependencies, this modularity is a key enabler for a better maintainable and evolvable language implementation.

5 Related work

In the survey of Jonne van Wijngaarden et. al. [15] current day transformation systems are compared and classified according to three transformation mechanics i.e. scope, direction and stage. Transformations with a wide target scope and a single node source scope are called local-to-global transformations. The survey explored whether the basic mechanisms like querying and traversing the parse tree are available in current day transformation systems in order to implement local-to-global transformations. In this paper we went a step further and investigated the repercussions of those mechanisms on the complexity, maintainability and evolvability of their implementations. We found that those mechanisms easily result into over-complex implementations and severely constrain maintainability and evolvability.

ASF+SDF [13] , Stratego [16], TXL [6], XSLT [2] provide a very powerful traversal system but the traversals to locate, collect, and redistribute the non-locals must be written against an intermediate parse tree consisting only of target nodes. Consequently the implementation easily results in a tangled web of traversals and rewrite rules as was discussed in section 2.2. The scoped dynamic rewrite rules of Stratego allow for non-local effects but their scopes are defined statically, in other words the scopes are determined at compile time. In our approach the scope of the integration rules are resolved at transformation time by the relocation process. Attribute grammars [8] and similar systems like intentional programming [10,11] have an implicit query system using inherited and synthesized attributes. The drawback of this mechanism is the lack of control over the source of the information, a feature which is vital for custom relocation strategies and for the specific integration of a non-local. Furthermore to prevent cylces, the implicit scheduling mechanism for the computations of attributes would require to decouple the production of the local from the non-local results. This decoupling has some drawbacks which were discussed and tackled in [16].

Although XSLT is equipped with a very powerful path language called XPath [1], it cannot be used to relocate non-local results. In our approach we are able to apply the XPath language for relocation, in a skinned and customized version of the language to ensure structure shyness.

Transformations in target driven transformation systems implicitly perform the integration process: they retrieve the necessary source nodes and use them to construct a target language construct. The source nodes are transformed, and integrated with one another using implicit correspondence and implicit combination rules between the generated parts. The approach we took on the other hand, made the integration explicit through separate integration rules which allowed the reuse and partial automation of rules and limited the impact of changes on the whole implementation.

In contrast to our approach none of the current day systems exploit the grammar to provide a semi-automatic relocation and composition system, and none of the systems provide an adaptable and modular integration mechanism that is driven declaratively.

6 Conclusion

Transformation systems, in particular rewrite rule systems, fail to deal with local-to-global transformations without corrupting maintainability and future evolvability. Our solution architecture enables the use of modular and simple rewrite rules to implement local-to-global transformations hereby avoiding complex, tangled traversals and rewrite rules and scheduling problems. We identified four identified coupling problems in current-day implementations. All of these have been reduced and some have been eliminated by our approach. Furthermore no other undesired couplings were introduced in the process. The impact of maintenance and future evolutions of one transformation to the implementation parts of that same transformation is localized and limited to specific modules. The relocation mechanism and the reusable integration, correspondence and combination rules across different implementations render the implementation of local-to-global transformation into a semi-automatic process, hence significantly reducing the implementation effort. This has a positive impact on the maintainability and future evolvability of the implementation.

References

[1] Anders Berglund, S. B., *XML Path Language (XPath) 2.0 W3C Working Draft 15 November 2002* (2002).

[2] Clark, J., *XSL Transformations (XSLT) Version 1.0 W3C Recommendation 16 November* (1999).

[3] Cleenewerck, T., *Component-based DSL Development*, in: *Proceedings of GPCE'03 Conference, Lecture Notes in Computer Science 2830* (2003), pp. 245–264.

[4] Cleenewerck, T., *Disentangling the Implementation of Local-to-Global Transformations in a Rewrite Rule Transformation System*, in: *Proceedings of the Symposium on Applied Computing Conference*, 2005.

[5] Cleenewerck, T., D. Derrider, J. Brichau and T. D'Hondt, *On the evolution of iMedia implementations*, in: *Proceedings of the European Workshop on the Integration of Knowledge, Semantics and Digital Media Technology*, 2004, pp. 173–180.

[6] Cordy, J., T. Dean, A. Malton and K. Schneider, *Software Engineering by Source Transformation - Experience with TXL*, SCAM'01 - Int. Workshop on Source Code Analysis and Manipulation (2001), pp. 168–178.

[7] Fabry, J., *But What if Things go Wrong? In Communication Abstractions For Distributed Systems workshop at ECOOP 2004* (2004).

[8] Knuth, D. E., *Semantics of context-free languages*, in: *Mathematical Systems Theory*, 1968, pp. 168–178.

[9] M., N. J., *The draco approach to constructing software from reusable components*, in: C. Rich and R. C. Waters, editors, *Artificial Intelligence and Software Engineering*, 1986, pp. 525–535.

[10] Simonyi, C., *The death of computer languages, the birth of Intentional Programming*, in: *The Future of Software, Proceedings of the Joint International Computers Limited/University of Newcastle Seminar*, University of Newcastle, 1995.

[11] Simonyi, C., *Intentional Programming - Innovation in the Legacy Age, IFIP WG 2.1 meeting, Microsoft Research* (1996).

[12] Smaragdakis, Y. and D. Batory, *Application Generators*, Encyclopedia of Electrical and Electronics Engineering (2000), j.G. Webster (ed.), John Wiley and Sons.

[13] Van Den Brand, M. and P. Klint, "ASF+SDF Meta-Environment User Manual," Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands (2002).

[14] Van Den Brand, M. G. J., P. Klint and J. J. Vinju, *Term rewriting with traversal functions*, ACM Trans. Softw. Eng. Methodol. **12** (2003), pp. 152–190.

[15] Van Wijngaarden, J. and E. Visser, *Program Transformation Mechanics*, Technical Report UU-CS-2003-048, Universiteit Utrecht (2003).

[16] Visser, E., *Stratego: A Language for Program Transformation Based on Rewriting Strategies*, Lecture Notes in Computer Science **2051** (2001), pp. 357–361.