A Cheap Implementation of Resugaring in BIRDS based on Bidirectional Transformation

Xing Zhang[1], Van-Dang Tran[4,2], and Zhenjiang Hu[3,2]

[1] Nankai University, China
[2] National Institute of Informatics, Japan
[3] Peking University, China
[4] The Graduate University for Advanced Studies, SOKENDAI, Japan

Abstract. Syntactic sugar refers to a certain syntactic structure added to the programming language. This syntactic structure has no effect on the function of the language, but is more convenient for programmers to use. Since syntactic sugar will be translated to the basic syntactic structure of the core language at the compilation stage, the relationship between the source program written with syntactic sugar and the execution of the core program is masked, and the compiled program is unfamiliar to programmers. It is not convenient for programmers to learn and debug source programs written with syntactic sugar. To solve that problem, this paper adopts the idea of resugaring for automatically transforming the evaluation sequence of the core language into the evaluation sequence of the surface language, and gives a cheap implementation using the existing bidirectional transformation tool BIRDS. The resugaring algorithms for both non-recursive and recursive desugaring transformations are implemented using Datalog, and the solutions to maintain two important properties of emulation and abstraction in the process of resugaring are studied.

Keywords: Desugaring · Resugaring · Bidirectional Transformation · BIRDS.

1 Introduction

Syntactic sugar plays an important role in extending a core language to a surface language with more user-friendly language constructs. The core language may be a small language with a simple syntactic structure but powerful functions, and it can be enriched to a surface language with additional syntactic sugars. A syntactic sugar can be defined by a set of desugaring rules, describing how it can be mapped to the core language. For instance, the following desugaring rule defines the syntax sugar Or, showing how to transform a surface term with Or to a core term with Let.

$$\mathrm{Or}([x, y, ys...]) \rightarrow \mathrm{Let}([\mathrm{Bind}(``t", x)], [\mathrm{If}(\mathrm{Id}(``t"), \mathrm{Id}(``t"), \mathrm{Or}([y, ys...]))])$$

This work is partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099.

One problem with syntactic sugars is that after the desugaring, the resulting programs (in the core language) become unfamiliar to programmers, and it obscures the relationship between the user's source program and the program being evaluated. To resolve this problem, the resugaring technique was proposed to lift the core evaluation sequence into one for the surface [17]. Given a surface term which can be desugared to a core term, resugaring is the process of adding syntactic sugars each step after the reduction of the corresponding desugared core term, in order to obtain an evaluation sequence expressed in surface language. When applied debugging and comprehension tools to core language terms resulting from desugaring, their output is also in terms of the core language. Resugaring can establish correspondence with the surface language that the user employs, so as to facilitate the use of those tools on the surface language[17]. Resugaring should satisfy the following two properties.

- **Emulation**. Each term in the generated surface evaluation sequence desugars into the core term which it is meant to represent.
- **Abstraction**. Code introduced by desugaring is never revealed in the surface evaluation sequence, and code originating from the original input program is never hidden by resugaring.

However, implementation of resugaring needs much effort. We need to implement both desugaring and resugaring carefully to make sure that they are consistent satisfying the emulation property. As pointed out in [17], this pair of transformations between terms of the core and the surface languages forms a bidirectional transformation; taking the surface program as the source and the core language as the view, then the forward transformation is desugaring, and the putback transformation is resugaring. This inspired us to consider a direct use of a bidirectional transformation language to implement resugaring (together with desugaring) so that the consistency between desugaring and resugaring is guaranteed for free. Furthermore, if we adopt to use a putback-based bidirectional language, where the forward transformation can be automatically derived from the putback transformation, by writing a resugaring program, a desugaring program can be automatically generated.

In this paper, we present a new implementation of resugaring using the putback-based bidirectional transformation tool called BIRDS (Bidirectional Transformation for Relational View Update Datalog-based Strategies) [20]. We use the BIRDS tool to develop the resugaring transformation, and automatically generate the corresponding desugaring transformation. Our main technical contributions are summarized as follows.

– We present a new implementation of resugaring using BIRDS based on bidirectional transformation. Our method is simpler and only needs to develop and maintain resugaring, which is in sharp contrast to the traditional method that needs to develop and maintain both desugaring and resugaring. Furthermore, our implementation satisfies the emulation property for free, and meets the abstraction property.

$$\begin{tikzcd} S \arrow[r, "get"] \arrow[d, bend left] & V \arrow[d, "u"] \\ S' & V' \arrow[l, "put"] \end{tikzcd}$$

Fig. 1. Bidirectional Transformation

– We show how syntax trees can be uniformly represented by relational tables, and how tree transformations (including pattern matching and substitution) can be efficiently implemented over relational operations in Datalog. This enables us to use BIRDS to implement a resugaring system, which integrates rule checking, desugaring and resugaring, and algebraic stepper [5, 6].
– Compared to the traditional implementation method, which uses about 1000 lines of codes in Haskell, our new method successfully implements the resugaring transformation using less than 200 lines of codes in Datalog for the same algorithm. Our method is powerful enough to deal with many difficult cases such as recursive syntactic sugars.

The remainder of this paper is organized as follows. After presenting some basic notions in Section 2, we propose an algorithm for performing resugaring using a view update strategy in Section 3. Section 4 shows how to maintain the emulation and abstraction in our implementation. Section 5 gives some examples, Section 6 summarizes related works, and Section 7 concludes this paper.

2 Preliminaries

2.1 Bidirectional Transformation

A bidirectional transformation (BX) [11] is a pair of a forward transformation *get* and a backward (putback) transformation *put* (see Fig. 1). The forward transformation *get* is a query over a source S that results in a view V. The putback transformation *put* takes the original S and an updated view V' as input to produce a new source S'. To ensure consistency between the source database and the view, *get* and *put* must satisfy the following round-tripping properties, called GetPut and PutGet:

$$\forall S, put(S, get(S)) = S \qquad \text{(GetPut)}$$

$$\forall S, V', get(put(S, V')) = V' \qquad \text{(PutGet)}$$

The GetPut property ensures that unchanged views correspond to unchanged sources. The PutGet property ensures that all view updates are completely embedded into the source such that the updated view can be computed again from the forward transformation *get* over the updated source.

```
1 % Schema declaration
2 source s1('X':int, 'Y':int).
3 source s2('X':int, 'Y':int).
4 view v('X':int, 'Y':int).
5 % View update strategy rules
6 -s1(X,Y) :- s1(X,Y), not v(X,Y).
7 -s2(X,Y) :- s2(X,Y), not v(X,Y).
8 +s1(X,Y) :- v(X,Y), not s1(X,Y), not s2(X,Y).
```

Fig. 2. A program in BIRDS

2.2 Bidirectional Programming with Datalog

We follow [20] and employ the BIRDS framework [1] to use the Datalog language with extensions for programming putback transformations, i.e. view update strategies. BIRDS [20] supports putback-based bidirectional programming [10, 13, 12]. In other words, the framework automatically checks the well-behavedness of a putback program written by the user and generates the corresponding forward (*get*) one for free. BIRDS further optimizes the user-written programs before compiling them into lower-level code.

Consider two base tables $s_1(X, Y)$ and $s_2(X, Y)$ and a view $v(X, Y)$, which is expected to be a union over s_1 and s_2. A program of view update strategy accepted by BIRDS consists of two essential parts: schema declaration and view update strategy rules. Figure 2 shows an example of the program, where s_1, s_2 and v are all binary relations with the same attributes `'X'` and `'Y'`. The first two rules (Lines 6 and 7) of the view update strategy say that if a tuple (X, Y) is in s_1 or s_2 but not in v, it will be deleted from s_1 or s_2, respectively. The last rule says that if a tuple (X, Y) is in v but in neither s_1 nor s_2, it will be inserted to s_1. BIRDS will automatically check the validity of the program and generate a view definition, i.e., forward transformation, as the following.

$$v(X, Y) :\!-\; s_1(X, Y).$$
$$v(X, Y) :\!-\; s_2(X, Y).$$

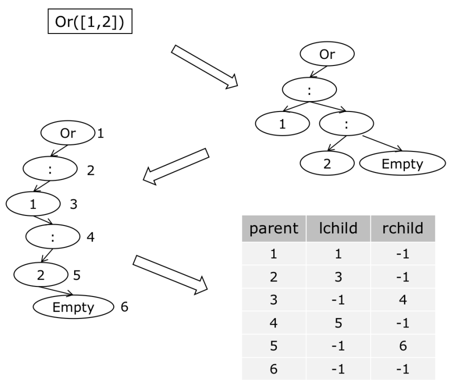
3 Resugaring as Putback Transformation

In this section, we shall explain how to write resugaring as a putback transformation (view update strategy), and how to use the BIRDS tool to run it and to automatically generate the corresponding desugaring program. We have implemented the complete putback program using Datalog.

3.1 Converting Syntax Trees to Relational Tables

Since BIRDS works only on relational database, the first step to use BIRDS is to convert syntax trees into relations (tables). This can be done straightforwardly,

https://github.com/nksezx/ResugaringAsPutback/blob/master/putback.dl



parent	lchild	rchild
1	1	-1
2	3	-1
3	-1	4
4	5	-1
5	-1	6
6	-1	-1

Fig. 3. Converting Syntax Trees to Relational Tables

because syntax trees can be considered as graphs, and graphs can be naturally represented by edge tables describing relationship between nodes. In order to facilitate the traversal and compress tables, we convert each abstract syntax tree into a binary tree, because the binary tree will have a fixed number of columns after it is expressed as a relational table.

As shown in Fig. 3, the syntax tree of Or([1,2]) is first converted into a binary tree. Then, we give each node a unique number and use the relational table to record the left and right child of each node. In this way, the relational table can be used to concisely represent syntax trees.

3.2 Designing Source and View Tables

To use BIRDS, we need to design source tables that represent the surface programs (surface terms) and transformation rules and the view tables that represent the core programs (core terms). A good relational table design can make the update strategy easier to write. In our system, we have the following five tables.

node. The node table records the information of all nodes in the patterns and terms. As shown in Table 1, *Id* is a unique key assigned to the created node in the semantic analysis. There are three types of nodes: keyword nodes, constant nodes, and variable nodes. There are three types of nodes in patterns (terms with variables) and only two in terms, which are keyword nodes and constant nodes. *Name* is the value of the node, which is a string. *Root* means the id of the root in the syntax tree.

pAst, sAst and cAst. These three tables are used to record the edge relationships of the syntax tree for patterns, surface syntax trees and core syntax trees. They are of the same schema as shown in Table 2: *Par* is the id of the

Table 1. The Node Table

id:number	name:string	type:string	root:number
1	Or/And/:	keyword	1
2	123/Empty/true	constant	1
3	x/y/z	variable	1

Table 2. The pAst Table

par:number	lchild:number	rchild:number
5	6	-1
6	-1	-1

parent node of each group, while *lchild* is the id of its left child and *rchild* is the id of its right child. *-1* means the node is missing.

The **pAst** table is to store all patterns. Consider the pattern of (Delay x). It is shown in Figure 4, where only two nodes *Delay* and *x* are included, and the ids of the two nodes are 1 and 2, respectively. The *x* node is the left child node of *Delay*, so we have (1, 2, -1) and (2, -1, -1) in Table 2. The cAst view and the sAst source respectively record the edge relationships in the core term and the generated surface term. And their fields are exactly the same as the pAst source, which represents the two patterns before and after transformation in all rules.

rule. The transformation rules are recorded in the rule table as shown in Table 3. The id of the root in the syntax tree uniquely represents the corresponding pattern. Let LHS represent the surface syntax pattern on the left side of the transformation rule, and RHS the core syntax pattern on the right side. Then, for the rule described in Figure 5, the id of the LHS is 1, and the id of the RHS is 5, so we store (1, 5) in rule table in Table 3.

3.3 Non-recursive Resugaring

Resugaring is to turn a core term back to a surface term by reversely applying transformation rules. Non-recursive resugaring refers to the case where the core term will be transformed back to at most one syntactic sugar and at most once.

Matching. Matching is to match the core term with the RHSs of the transformation rules, so as to obtain the rules used in the resugaring and the mapping relationship between variables in patterns and bindings in the core term.

There are three types of nodes in the pattern, but there are only two types in the term. Therefore, after a simple combination, there are four cases (in Table 4) that meet the matching conditions. If the types of the pattern node and the term node are constants or keywords, then the matching succeeds when the names of the two are the same. If the pattern node is a variable, then the term node can be a constant node or a keyword node, because the variable can match a constant or sub-expression.

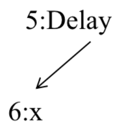


Fig. 4. A pattern

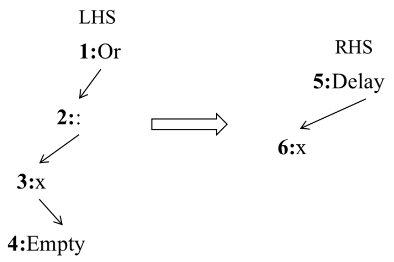


Fig. 5. A Transformation Rule

Table 3. rule source

lhs:number	rhs:number
1	5

The matching algorithm consists of three steps: the first step is to recursively try to match starting from the root of the core term and the root of all RHS patterns; the second step is to determine the RHS that the core term completely matched, thereby we can determine the transformation rule that can be used; the third step is to extract the information needed for the subsequent substitution algorithm. Algorithm 1 describes the matching algorithm in detail with a specific example. The core term and related transformation rule are shown in Figure 6.

As shown in Figure 7, when the root of the term matches the root of the RHS, we will continue matching their left child and right child respectively, so the first step of matching is a recursive process. In the second step, we found that all the nodes of the above RHS were successfully matched to the term node in the first step, so we think that the matching was successful. So in the third step, we determined the corresponding LHS and the bindings corresponding to the variables in the LHS, namely $\{x \to 1,\ y \to 2\ \ ys \to Empty\}$.

Substitution Substitution refers to the process of replacing the variables in the LHS determined in the matching with the corresponding bindings to obtain the final surface term.

3.4 Recursive Resugaring

Recursive resugaring refers to the case where the core term will use multiple syntactic sugars.

Matching Sub-term is a collection of partial nodes in the binary tree of the core term. These nodes can reach each other through their common edges. The meta term is a special sub-term, and its nodes can be bijective with all the nodes of a certain pattern. Therefore, the object of recursive resugaring is a core term

Table 4. Matching Conditions

Pattern Node	Term Node
keyword	keyword
constant	constant
variable	keyword
variable	constant

Algorithm 1 Matching Algorithm

```
def matching(CoreTerm)
    // step 1
    foreach RHS in RHSs
        PROOT = Root ID of RHS
        TROOT = Root ID of CoreTerm
        trymatch(PROOT, TROOT)
    // step 2
    check()
    // step 3
    insert rule which RHS is 'completely matched' into 'state'
    if size of 'state' == 0:
        Raise('match failed!')
        return
    foreach (PID, TID, RHS) in 'matched':
        if RHS is 'completely matched' and type of PID is 'variable'
            insert (PID, TID) into 'env'
    foreach TID exists in 'env'
        insert subtree of TID into 'value'
```

with multiple meta terms. The core idea of the recursive transformation algorithm is to match each sub-term in parallel, so the matching should not only start from the root of the binary tree of the core term, but should try from each node. Starting from a certain node, the part of the nodes that successfully matches a pattern is a meta term.

The following uses an example to explain the recursive resugaring process in detail. As shown in Figure 8, The nodes of the three colors of blue, gray and orange are the meta terms matching three different patterns respectively. We no longer only match from the root If, but each node of the core term can be used as the root to match with roots of patterns. From this, we can determine that the following rules will be used in transformation.

And([x, y, ys ...]) → If(x, And([y, ys ...]), False)
Or([x, y, ys ...]) → Let([Bind("t", x)], [If(Id("t"), Id("t"), Or([y, ys ...]))])
If(x,y,z) → If(x,y,z) (a special rule, see 4.2)

Expansion In Matching, we can determine the patterns to be converted from the patterns which meta terms successfully matched. In the expansion, we combine the determined patterns according to the positional relationship suggested

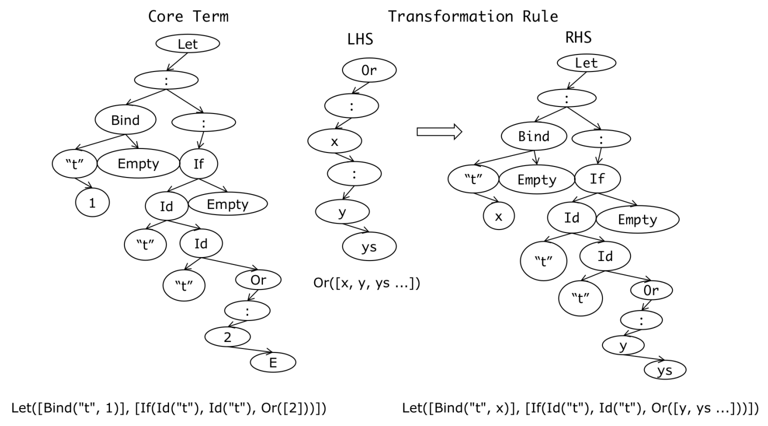

Core Term
Transformation Rule
LHS
RHS
Or([x, y, ys ...])
Let([Bind("t", 1)], [If(Id("t"), Id("t"), Or([2]))])
Let([Bind("t", x)], [If(Id("t"), Id("t"), Or([y, ys ...]))])

Fig. 6. A Core Term and A Transformation Rule

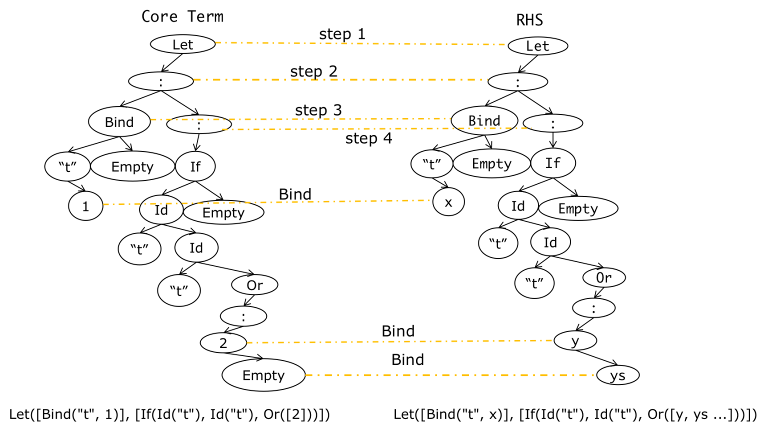

Core Term
RHS
step 1
step 2
step 3
step 4
Bind
Bind
Bind
Let([Bind("t", 1)], [If(Id("t"), Id("t"), Or([2]))])
Let([Bind("t", x)], [If(Id("t"), Id("t"), Or([y, ys ...]))])

Fig. 7. Matching

by the core term into a combined pattern. In the previous example, we combined the target patterns as shown in Figure 9.

Substitution The substitution process here is consistent with non-recursive resugaring which replaces variables in the combined pattern with bindings.

4 Two Properties

4.1 Checking Transformation Rules for Preserving Emulation

Emulation means that the surface terms obtained by resugaring maintain the same semantics as the corresponding core terms. In other words, the surface term obtained after resugaring can be desugared to the corresponding core term again.

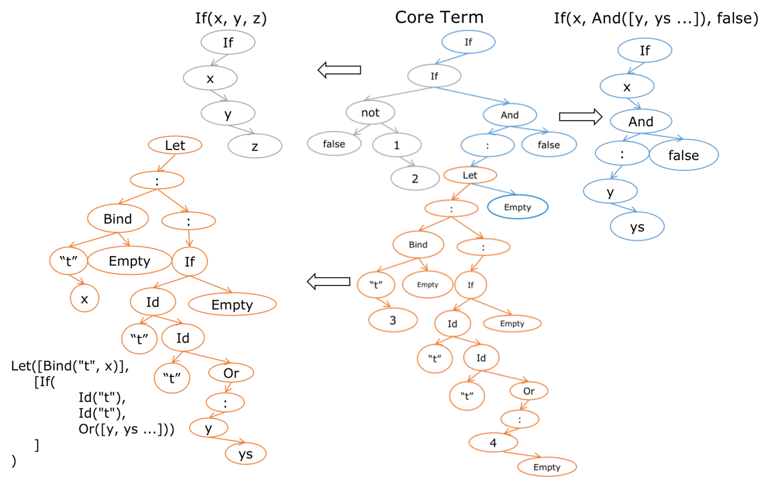


Fig. 8. Matching

In fact, this is also the PutGet property that the bidirectional transformation program needs to satisfy. Therefore, BIRDS can ensure that a single transformation rule satisfies emulation, but cannot guarantee the situation of multiple transformation rules. Some transformation rules will violate the emulation. For example, we use the following transformation rules to resugar Max([$-\infty$]).

$$\begin{aligned} &\text{Max}([\,]) &&\rightarrow \text{Raise}(\text{“empty list”}); \\ &\text{Max}(xs) &&\rightarrow \text{MaxAcc}(xs, -\infty); \end{aligned}$$

The following core language evaluation sequence will be obtained.

$$\text{MaxAcc}([-\infty], -\infty) \Rightarrow \text{MaxAcc}([\,], -\infty)$$

After adding sugar to them, we will obtain the following surface language evaluation sequence:

$$\text{Max}([-\infty]) \Rightarrow \text{Max}([\,])$$

But Max([ ]) should be desugared to the error message, instead of the second step of the core language evaluation sequence. But with the following transformation rules, the above problems will not occur.

$$\begin{aligned} &\text{Max}([\,]) &&\rightarrow \text{Raise}(\text{“empty list”}); \\ &\text{Max}(x : xs) &&\rightarrow \text{MaxAcc}([x, xs, ...], -\infty) \end{aligned}$$

Therefore, we need to check whether there is any overlap among the transformation rules. If there is any overlap, the rules are invalid, otherwise they are valid. In Figure 10, the first group has overlap, but the second group does not have any overlap.

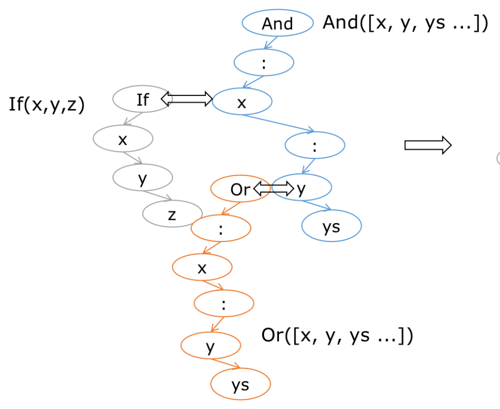


Fig. 9. Expansion

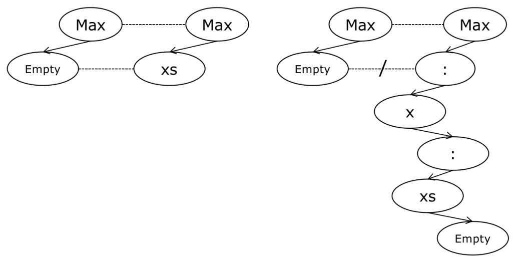


Fig. 10. Match the LHSs with Each Other

4.2 Maintaining Abstraction

Abstraction means that the code obtained by desugaring will not be leaked in the output surface language program, and at the same time, the program in the original input cannot be resugared. If the surface term itself uses some core language syntax, such as ”If”:

$$\text{Let}([\text{Bind}(\text{“}t\text{”}, not(false))], [\text{If}(\text{Id}(\text{“}t\text{”}), \text{Id}(\text{“}t\text{”}), \text{Or}([true]))]).$$

After reduction, we hope that it will not be converted during resugaring, that is, Or ([not(true), true]) cannot be obtained.

Our solution is to treat the core syntax rules as special transformation rules with the same LHS and RHS. We mark the roots that match the core syntax rules with a “fixed” label, and the stepper will retain the labels during the reduction. In the above example, since this surface language will match core syntax rule of *Let*, we add “fixed” tag in front of *Let*. Then, in resugaring, the marked nodes will not be used to match with RHSs. That is to say, the above core term will not match the RHS of Or syntactic sugar, so we can maintain abstraction during transformation.

Num	Surface Language Evaluation Sequence	Core Language Evaluation Sequence
1	And([If(not(false), 1, 2), Or([3, 4])])	If((Tag Fixed If(not(false), 1, 2)), And([Let([Bind(“t”, 3)], [If(Id(“t”), Id(“t”), Or([4]))])]), false)
2	And([If(true, 1, 2), Or([3, 4])])	If((Tag Fixed If(true, 1, 2)), And([Let([Bind(“t”, 3)], [If(Id(“t”), Id(“t”), Or([4]))])]), false)
3	And([1, Or([3, 4])])	If(1, And([Let([Bind(“t”, 3)], [If(Id(“t”), Id(“t”), Or([4]))])]), false)
4	/	And([Let([Bind(“t”, 3)], [If(Id(“t”), Id(“t”), Or([4]))])])
5	Or([3,4])	Let([Bind(“t”, 3)], [If(Id(“t”), Id(“t”), Or([4]))])
6	/	If(3, 3, Or([4]))
7	3	3

Fig. 11. Resugaring

5 An Example

We show an example of our resugaring. As shown in Figure 11, except for the fourth and sixth steps, other core terms have the corresponding surface terms after resugaring, and satisfy emulation and abstraction. For example, the second step of the core language evaluation sequence is the result after *not(false)* is evaluated as *true*. Resugaring adds *And* and *Or* syntactic sugar to the second step of core to get the second step of surface.

6 Related Work

Bidirectional transformation (*bx*) is a mechanism used to maintain the consistency of two (or more) related sources of information. Researchers from many different fields, including software engineering [18, 2], programming languages [7, 15], databases [3, 4] and document engineering, are actively studying the use of *bx* to solve various problems [8]. Our work shows the application of *bx* in programming language transformation.

The view update problem is a classical problem that has a long history in database research. A typical language of the putback-based approach is BiGUL [14, 12], which supports programming putback functions declaratively while automatically deriving the corresponding unique forward transformation. Based on BiGUL, Zan et al. [21] design a putback-based Haskell library for bidirectional transformations on relations. But only [20] can run in database environments, which provides us with great convenience for language transformation on relational tables.

Data interoperability addresses the ability of systems and services that create, exchange and consume data to have clear, shared expectations for the contents, context and meaning of that data [9]. Because our approach solves the problem of correlation between the surface language and the core language being executed, the above problem is a typical problem of maintaining data interoperability. Many methods have been proposed to solve the problem. One method is to manually redefine the semantics based on the surface language, which undermines the benefits provided by the small core. The other is to use source tracking [19], but this is actually not a solution: users will still only see the core language after desugaring. In addition, creating a one-time solution for a given language is not suitable for those languages where users can create other syntactic sugar in the program itself, such as Lisp language [16].

Resugaring [17] is currently the most effective method. The traditional implementation method is to write two programs, namely, desugaring and resugaring, and then use string processing to transform the programming language. And we use the BIRDS [20] to automatically generate the desugaring program through the resugaring program, and convert the processing of the string into the operation of the relational table.

7 Conclusions

In this paper, we propose a new implentation method for resugaring, which can greatly reduce the difficulty of implementation. Not only did we propose to use the bidirectional transformation tool BIRDS to write the resugaring algorithm, but also presented a new solution to maintain emulation and abstraction during the transformation.

In the future, firstly, we intend to generalize the transformation algorithms on syntax trees by using more general bidirectional transformation on trees. Secondly, the tree is a special case of graph, and the relational table operation for the graph pattern and the tree pattern is not much different, so we intend to generalize the algorithm to the bidirectional transformation of the graph pattern.

References

1. BIRDS. https://dangtv.github.io/BIRDS/
2. Antkiewicz, M., Czarnecki, K.: Design Space of Heterogeneous Synchronization, pp. 3–46. Springer Berlin Heidelberg (2008)
3. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6**(4), 557–575 (Dec 1981)
4. Bohannon, A., Pierce, B., Vaughan, J.: Relational lenses: A language for updatable views. pp. 338–347 (01 2006)
5. Clements.: Portable and high-level access to the stack with Continuation Marks. Ph.D. thesis, Northeastern University (2006)

6. Clements, J., Flatt, M., Felleisen, M.: Modeling an algebraic stepper. In: Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2028, pp. 320–334. Springer (2001)
7. Culpepper, R., Felleisen, M.: Debugging hygienic macros. Science of Computer Programming **75**(7), 496 – 515 (2010), generative Programming and Component Engineering (GPCE 2007)
8. Czarnecki, K., Foster, N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional transformations: A cross-discipline perspective. vol. 5563, pp. 260–283 (06 2009)
9. Dell'Erba, M., Fodor, O., Ricci, F., Werthner, H.: Harmonise: A Solution for Data Interoperability, pp. 433–445. Springer US (2003)
10. Fischer, S., Hu, Z., Pacheco, H.: A clear picture of lens laws - functional pearl. In: Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1. vol. 9129, pp. 215–223. Springer (2015)
11. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14. pp. 233–246 (2005)
12. Ko, H.S., Hu, Z.: An axiomatic basis for bidirectional programming. Proc. ACM Program. Lang. **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158129
13. Ko, H., Zan, T., Hu, Z.: Bigul: a formally verified core language for putback-based bidirectional programming. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 61–72 (2016)
14. Ko, H.S., Zan, T., Hu, Z.: Bigul: A formally verified core language for putback-based bidirectional programming. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. p. 61–72. PEPM '16 (2016)
15. Krishnamurthi, S., Erlich, Y.D., Felleisen, M.: Expressing structural properties as language constructs? In: Programming Languages and Systems. pp. 258–272. Springer Berlin Heidelberg (1999)
16. Lapalme, G.: Implementation of a "lisp comprehension" macro. SIGPLAN Lisp Pointers **IV**(2), 16–23 (Apr 1991). https://doi.org/10.1145/121983.121985
17. Pombrio, J., Krishnamurthi, S.: Resugaring: lifting evaluation sequences through syntactic sugar. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 361–371. ACM (2014)
18. Schürr, A.: Specification of graph translators with triple graph grammars. In: Graph-Theoretic Concepts in Computer Science. pp. 151–163. Springer Berlin Heidelberg (1995)
19. Tirkel, A., Rankin, G., van Schyndel, R., Ho, W., Osborne, C.: Electronic watermark (12 1993)
20. Tran, V.D., Kato, H., Hu, Z.: Programmable view update strategies on relations. PVLDB **13**(5), 726–739 (2020)
21. Zan, T., Liu, L., Ko, H.S., Hu, Z.: Brul: A putback-based bidirectional transformation library for updatable views. In: Bx@ETAPS (2016)