An Overview on Bidirectional Transformations

José Nuno Macedo
Department of Informatics
University of Minho
Braga, Portugal
ze_nuno_eu@hotmail.com

***Abstract*—Bidirectional transformations are a way to maintain consistency between various related sources of information. From Model-Driven software, to relational databases and Domain-Specific Languages, there are various applications of this technology that provide a stable and reliable methodology and set of tools to solve problems in these areas.**

This report presents bidirectional transformations, providing an insight into this methodology. Some approaches to practical software development using this technique are also described.

***Index Terms*—bidirectional transformation, synchronization, structured programming, model update**

I. Introduction

Software development has evolved considerably over the last decades. The problems presented have become increasingly complex, with increasingly complex solutions arising to combat them. Open-source development, several frameworks, alternative programming languages and paradigms and software testing are some solutions for the clutter and confusion that complex projects can create, bringing some stability to the chaos of complex software. Some of these tools, such as frameworks, help by guiding the developer in the right direction, possibly being more restrictive while doing so. Some improve development experience by providing tools to detect errors or oversights that can and will happen in large projects, but can be attenuated through collaborative development and the use of software testing techniques.

While most of these tools help the project become more robust, the availability of formally proven tools is still lackluster - there is no mathematical approach that can, for most frameworks and software testing, guarantee correctness. Therefore, several times, the security provided by these tools is an illusion, allowing the developer to be more daring in their approach with no scientific backing.

Several problems in software engineering are based on maintaining several data structures that must be consistent in their behaviour and content. Consistency between the several data structures can be provided by a third-party program, thus increasing the burden on the developer due to the need to develop more software to complement the original intention. This can be entirely avoided by using an approach that deals with these consistency needs automatically. Bidirectional transformations are a mechanism to improve the development around such problems.

This reports aims to introduce bidirectional transformations as a concept, supported by some examples and the description of practical approaches to this concept. In section II, the concept of bidirectional transformations is introduced, along with some examples that illustrate possible applications of this technique. In section III, a practical approach, known as lenses, is approached. In section IV, *BiGUL*, a putback-based bidirectional programming language, is briefly introduced. Section V concludes this overview, pointing to more relevant existing work.

II. Bidirectional Transformations

A bidirectional transformation model (from now on referred as BX) contains different representations of shared data. When any representation of the shared data is modified, all of the representations are modified as well, reflecting this change accordingly. This results in a permanently synchronized environment, where every representation of data is consistent with the others. In this report, the focus will be set on the binary case, that is, when two different representations of data need to be synchronized. This is a general case present in various software projects, and BX is applicable in most of them.

A. Converting Data

A BX is an interesting approach for data conversion. Assuming there are two different representations of the same data that need to be consistent, a BX can be used to maintain the consistency. An example of this is a code editor, with syntax checking and automatic correction of common errors. In this example, there are two different representations of the code written by a user. The most obvious is the textual representation, that is, the code that the user edits. There is, however, a different representation, which is the internal representation of said code.

This internal representation is needed as there can be a lot of redundant and useless information in the code the user writes. For example, white spaces, indentation and comments are typically all useless in terms of the actual program being written. Of course, a code editor might want to not ignore comments and format them appropriately, but the fact that some redundant information remains is still valid. At the same time, the internal representation is usually in a different format than the external one. Since the code is being parsed, it is expected that an *Abstract Syntax Tree* is being generated. This is a tree where each node represents an instruction or segment of the program, being fundamentally different from the external representation as the data is not a simple

string anymore, being instead an appropriate data type for the information stored. Figure 1 represents the behaviour of the code editor.

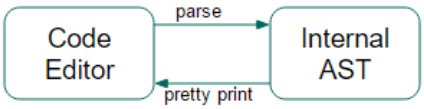

Fig. 1. How the code editor behaves.

When an user writes code in the code editor, it is expected that the code editor will, in turn, detect any mistakes and produce a warning message. When this happens, a change that was applied in the external representation must be reflected appropriately in the internal representation so as to maintain consistency. If consistency is not kept, it can be possible that an error in the code is not flagged by the code editor, or that a correct line of code is flagged as incorrect. Unless the code editor regularly checks for consistency faults or rebuilds the whole structure, the two data representations will stop being synchronized, resulting in an unpleasing experience for the user, where the software they use sabotages the workflow instead of providing valuable feedback.

Typically, this type of applications are developed using two separate tools, one for each direction, that is, in this example, a *parser* and a *pretty printer* are developed separately, and then integrated into the main software. A BX can simplify the development process of such tools by providing a consistent approach for this development. Some tools for BX development also provide security in terms of assuring the synchronization of the generated BX.

B. Database views and updates

The view-update problem on databases [1] was one of the earliest problems to be studied as a BX, albeit the terminology used was different. For this problem, consider an extremely complex database representing the employees of a company, and the corresponding projects of each. There are various ways of consulting such database, some more complex than others. It is possible to look at one table individually, to look at several tables one at a time, to join various tables that share common values.

In this example, an *Employees* table contains the name and details of the employees of the company. Naturally, a big volume of information is contained in this table, such as address, phone number, e-mail, accounting details, among others. A very simplified version of this table is presented in Table I.

In the same database, another table, designated the *Projects* table, contains information relative to projects which are being developed at said company. Examples of data that can be found in this table include budget, expected finish date, starting date, infrastructure, employees involved, status of the project. A very simplified version of this table is presented in Table II.

TABLE I
EMPLOYEES TABLE

Id	Name	Age
1	John	32
2	Jack	27

TABLE II
PROJECTS TABLE

Id	Employee	Status
1	1	Review
2	1	Evaluation

By joining the two tables, a new, more complete table, is created. This table is created by uniting the two previous tables, matching the lines where the employee *ID* is equal. In it, it is easier to see which employee is working on which project. Thus it can be concluded, on Table III, that John is working in projects 1 and 2.

TABLE III
UNION OF THE EMPLOYEES AND PROJECTS TABLES

Project_Id	Employee	Status	Age
1	John	Review	32
2	John	Evaluation	32

Table III can be read as a *view* of the *source*, which is the entire database, containing tables I and II. For complex databases, a view is necessary to generate a readable table, as various tables have to be connected in order to form a readable result. However, this view poses a problem.

If the user performs any type of changes in the view, then those changes need to be immediately reflected on the database accordingly. This is because the consistency must be kept, but, more important than that, because otherwise the changes the user performs could be lost.

In this example, if the user deletes the name *John* from both entries, there is no well-defined behaviour for the BX. In the *Projects* table, projects could be kept with null entries for the employees or the projects could be kept and the status changed to *Cancelled*. In the *Employees* table, *John* could be removed, representing that he was fired, or he could be kept, and perhaps relocated to a different project or section of the company.

The correct behaviour must be specified beforehand, in a way that assures consistency between user actions. The propagation of an update from the view to the source must also be correct, such that it does not destroy the database in any way.

C. Development based on Software Models

When developing software, a possible approach is to create a model representing the part of the software that is to be worked on. As such, a layer of abstraction is created, and the developer can focus on just the important aspects, present in

the model, while abstracting the irrelevant aspects, which are hidden by the model. In this situation, BX are relevant as the model is a view of the original software, which is the source. As such, changes applied to the model should be reflected on the original software appropriately, and vice versa.

A software product can have several different models related to it. They can be different representations of the same component, or different components. However, the purpose is always to abstract. The updates to be performed on the original software when a model is changed can be extremely complex in some cases, as some models, through abstraction, create various layers of complexity in the process of actually applying the changes to the original software.

One of the most commonly known examples where this can be applied is the object-relational mapping (ORM) technique. According to it, in object-oriented programming languages, the software should not access the database directly, opting instead for having a layer for communication with the database. This layer contains various objects that represent parts of the database, and all of the communications with it are described inside, such that a neat and clean interface is provided for the communication. In a sense, these objects are models of parts of the database. The developer does not have to be concerned with how the database works or responds, he only has to interact with the simple-by-design objects.

III. Lenses

A. Definition

While there are several techniques for approaching BX in terms of development, this report will shift the focus into the lenses approach. This is an approach that has gained a lot of traction in recent years [2] [3] [4]. An asymmetrical lens consists of two functions, given a source S and a target V:

$$get : S \to V \tag{1}$$
$$put : S \times V \to S \tag{2}$$

The *get* function, given a source, produces a corresponding view, such that *get s* would produce a view *v*. The *put* function, given the old source and an updated view, updated the source with the changes applied to the view, such that *put(v',s)* would produce a new, updated source. It is asymmetrical due to the fact that the view determined by the source.

For the database example, a view can be a certain join of two tables, obtainable by a *get* function. The *put* function then updates the database with the updated table.

Additionally, a lens is *well-behaved* when it satisfies:

$$\forall s \quad put(s, get\ s) = s \tag{3}$$
$$\forall s, v \quad get(put(s, v)) = v \tag{4}$$

The law 3 is known as the *GetPut* law. It states that putting a view as it was taken from the source does not change it. This is a desired behaviour in various applications of BX- it is expected that, if a joined table from a database is unchanged, then the database itself should remain unchanged. This property is sometimes referred as a hippocratic property, that is, a property that prevents unnecessary harm in the system.

The law 4 is known as the *PutGet* law. According to it, when putting a view into the source, and then taking a view from the source, the resulting view should be equal to the original view. In some situations, it is desired, in others, not so much. Putting an invalid view into the database, for example, can result in ignoring the put command, and therefore performing a get afterwards will yield different results. In the database example, inserting an employee with a negative age can be considered an invalid put, and therefore discarded. However, when assuming that the view to put into the source is valid, then this property is also generally desired. It is sometimes referred as a correctness property, as it guarantees that the put action is performing desired results.

A well-behaved lens is *very well-behaved* when it satisfies:

$$\forall s, v, v' \quad put(put(s, v), v') = put(s, v') \tag{5}$$

The property 5 is known as the *PutPut* property. According to it, putting a view into the source and then immediately putting another view in the resulting has the same effect as only putting the last view into the source. This happens when putting a new view overwrites the results of putting a previous view.

There are other types of lenses besides symmetrical lenses, such as symmetric lenses [5], edit lenses [6] and matching lenses [7].

B. Practical Approach

There have been various practical approaches to BX systems in programming over the last few years. The focus, however, will be set in a lens library for the Haskell programming language [8]. This library provides a set of tools for building and manipulating lenses in Haskell. This set of tools allows the developer to very easily manipulate complex data structures, by simplifying the access of these data structures through lenses.

In practice, this library provides ways to build *put* and *get* functions adequately for each problem. In Listing 1, several examples of code are presented.

Listing 1. Examples of lenses using this library

```
source = ("hello",("world","!!!"))

get_view1 = ^._1
get_view2 = ^._2._1

view1 = get_view1 source
view2 = source^._2._1

put_view1 = set _1 42
put_view2 = set (_2._1) 42

update1 = put_view1 source
update2 = set (_2._1) 42
            ("hello",("world","!!!"))
```

In this block of code, a source is declared, according to the type $S \times (S \times S)$, where *S* is a text string. In fact, this is a simple source, but it is used for demonstration purposes. Two *get* functions are defined, namely *get_view1* and *get_view2*. The ^ operator indicates a specification of a *get*, and the following argument is the position of the record that is related to this operation. In *get_view1*, *_1* is used to specify the first record, and in *get_view2*, *_2._1* is used to specify the first record inside the second record.

The *view1* and *view2* constants represent views, calculated from applying the *get* functions to the source. Two alternative but equivalent definitions are shown in these definitions. The constant *view1* contains the first record, that is, *"hello"*, while *view2* contains the first record inside the second record, that is, *"world"*.

For the definition of *put*, two examples are also displayed. The function *put_view1* puts the value *42* in the first record, and the function *put_view2* puts the value *42* in the first record inside the second record. As such, applying *put_view1* onto the source produces a new source of value *(42,("world","!!!"))*, and applying *put_view2* onto the source produces a new source of value *("hello",(42,"!!!"))*.

It is important to note that Haskell is a strongly-typed programming language. As such, it is not always trivial to change the type of the data in a simple way. However, the lenses provided in this library are flexible in this sense, as they allow the developer to either create simple lenses that preserve the type of information, or slightly more complex lenses that can modify the type of information while still abiding to the same rules and operators that are used for the simple lenses. In fact, lenses can also be composed, and as such, a complex lens for a complex source can be defined as a composition of various simple lenses, which are easy to understand and debug individually, but act as building blocks of a powerful tool.

For more complex data types, this library allows for the use of a Template Haskell construct to automatically derive adequate lenses. In practice, this results in efficient development cycles for the developer, as manually creating lenses for complex data structures can be time-consuming and confusing. This library is open-source and well documented, with hundreds of examples fit to various different problems.

IV. BiGUL

Putback-based bidirectional programming is an approach to bidirectional programming in which part of the development cycle is automatically generated by construction. In fact, it defines that, for a BX to be properly defined, only the putback, that is, the *put* function, must be defined, and the *get* function can be automatically derived from it. The opposite is not true - given a *get* function, there are several ways to define the *put* function.

Considering the previous example of the database, when a view, that is, a joined table, is changed, by deleting an employee, there are several ways to reflect that change on the database. However, given a definition on how to reflect the change on the database, the reverse process, that is, getting the information from the database, is unique.

BiGUL (Bidirectional Generic Update Language) [9] is a putback-based bidirectional programming language, complementar to the Haskell programming language. It is formally verified with the Agda [10] programming language, therefore guaranteeing that any putback transformation written in BiGUL is well-behaved, that is, that they obey the prepositions 3, 4 and 5.

While *BiGUL* is an elaborate tool that allows for the development of complex and deep systems, it is not as user-friendly as the lenses library presented in subsection III-B. As such, this overview shall only cover some constructs present in it briefly, while still encouraging readers to read upon and experiment with this tool [11].

The following constructs are building blocks available in the *BiGUL* language, for the definition of *put*:

- ***Replace*** - Replaces a value in the source with the provided value in the view.
- ***Skip*** - Ignores the value in the view, thus keeping the source intact.
- ***Fail*** - Fail, producing an error message. Useful for defining incorrect behaviour.
- ***CaseS*** - Produce a case statement, similar to the *switch* statement in several imperative languages, where the source is matched against conditions until one is matched, and then the associated code is executed.
- ***CaseV*** - Similar to *CaseS*, but matching against the view.
- ***RearrS*** - Rearrange the source into a more desirable intermediate data representation. Useful for facilitating some more complex matches between structures that differ between the source and view.
- ***RearrV*** - Similar to *RearrS*, but rearranging the view.
- ***Align*** - Match a list of information in the view with a more complex list of information in the source, therefore describing how to perform a *put* into a more complex structure.
- ***Update*** - Syntactic sugar, allows the use of pattern matching to simplify the usage of some of the previous constructs.

Having these constructs defined, complex *put* operations can be defined by composing them adequately, according to the syntax of *BiGUL*. By using them, the developer has a formal guarantee that the BX is well-behaved, as well as only having to specify part of the program and having the rest be generated "for free".

The developer can also force the usage of a self-defined *put*, instead of using the available tools to build one. While this may allow for an easier definition of *put*, it also means that there is no formal backing or automatic generation of *get*, so the advantages of the tool are lost. However, if there is a guarantee that the supplied *put-get* pair is correct, then this approach can facilitate the development of some programs where some components can be difficult to express in *BiGUL*, and therefore be expressed directly instead.

V. Conclusions

In this report, bidirectional transformations are presented as an underlying problem to several software projects, and at the same time, as the solution. In fact, thinking of software problems as bidirectional transformations can provide more insight on how to adequately approach them, using techniques that are safe and simple to use when correctly employed.

Some contextualization is provided through the data conversion, database updates and software model development examples. However, it is important to keep in mind that bidirectional transformations are not stuck in these areas - these are merely examples of a more generic approach to synchronization problems.

Contextualized into bidirectional transformations, the lenses approach is a recent and interesting approach to solving complex problems. In fact, it is an abstract concept, which has been implemented into, among others, the library described in subsection III-B.

BiGUL, on the other hand, implements the concepts of bidirectional transformations into a concrete programming language, supported by formal backing and automated program generation. As opposed to the lenses library, which is a complementary library for the Haskell programming language, *BiGUL* describes a standalone language.

There is plenty of work in the area of bidirectional transformations, ranging from work on XML Schemas [12] to applications of this technique to databases. The refinement of this approach represents an important step in the evolution of synchronization approaches for software problems, as it proposes interesting approaches to problems that are relatively common in software development but not always correctly handled.

References

[1] F. Bancilhon and N. Spyratos, "Update semantics of relational views," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 557–575, Dec. 1981. [Online]. Available: http://doi.acm.org/10.1145/319628.319634

[2] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, May 2007. [Online]. Available: http://doi.acm.org/10.1145/1232420.1232424

[3] A. Bohannon, B. C. Pierce, and J. A. Vaughan, "Relational lenses: A language for updatable views," in *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '06. New York, NY, USA: ACM, 2006, pp. 338–347. [Online]. Available: http://doi.acm.org/10.1145/1142351.1142399

[4] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt, "Boomerang: Resourceful lenses for string data," in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08. New York, NY, USA: ACM, 2008, pp. 407–419. [Online]. Available: http://doi.acm.org/10.1145/1328438.1328487

[5] M. Hofmann, B. Pierce, and D. Wagner, "Symmetric lenses," *SIGPLAN Not.*, vol. 46, no. 1, pp. 371–384, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925844.1926428

[6] ——, "Edit lenses," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 495–508. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103715

[7] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce, "Matching lenses: Alignment and view update," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '10. New York, NY, USA: ACM, 2010, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/1863543.1863572

[8] E. A. Kmett, "Lenses, folds and traversals," https://github.com/ekmett/lens, 2018.

[9] H.-S. Ko, T. Zan, and Z. Hu, "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*, ser. PEPM '16. New York, NY, USA: ACM, 2016, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2847538.2847544

[10] U. Norell, "Towards a practical programming language based on dependent type theory," 2007.

[11] Z. Hu and H.-S. Ko, *Principles and Practice of Bidirectional Programming in BiGUL*. Cham: Springer International Publishing, 2018, pp. 100–150. [Online]. Available: https://doi.org/10.1007/978-3-319-79108-1_4

[12] Z. Hu and J. de Lara, Eds., *Theory and Practice of Model Transformations - 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7307. Springer, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-30476-7