Bidirectional Spreadsheet Formulas

Nuno Macedo*, Hugo Pacheco†, Nuno Rocha Sousa*, Alcino Cunha*
*HASLab, INESC TEC & Universidade do Minho, Portugal {nfmmacedo,nrsousa,alcino}@di.uminho.pt
†Cornell University, USA {hpacheco}@cs.cornell.edu

***Abstract*—Bidirectional transformations have potential applications in a vast number of computer science domains. Spreadsheets, on the other hand, are widely used for developing business applications, but their formulas are unidirectional, in the sense that their result can not be edited and propagated back to their input cells. In this paper, we interpret such formulas as a well-known class of bidirectional transformations that go by the name of lenses. Being aimed at users that are not proficient with programming languages, we devote particular attention to the seamless embedding of the proposed bidirectional mechanism with the typical workflow of spreadsheet environments, allowing users to have a fine control and understanding of the behavior of the derived backward transformations.**

I. Introduction

Transforming data between two different domains is a typical problem in software engineering. Ideally such transformations should be *bidirectional*, so that changes in either domain can be propagated to the other one. Many existing bidirectional transformation (BX) frameworks are instantiations of so called *lenses* [1], a BX programming paradigm that tackles the well-known view-update problem: given a function that queries specific information from a source domain, how can modifications to the view be translated back to the original source? Despite the high demand and recognized potential of BXs, they have still not found the desired adoption among commercial products (with [2] as a notable exception).

Spreadsheets are one of the most popular tools for storing and processing information, with more than 500 million estimated users worldwide, providing a semi-structured data model and a simple programming model that make initiation easier for non-proficient users [3]. At the heart of spreadsheet programming are formulas, that automate calculations by describing how the value of a target cell can be calculated from other cells. Spreadsheet formulas are commonly used to transform entities like dates/phone-numbers from one format to another, perform arithmetic computations, compute summary reports, etc. Spreadsheet systems like Microsoft Excel or Google Spreadsheets provide rich libraries with a myriad of numerical and string manipulation functions, as well as lookup functions for the manipulation of arrays or matrices of cells.

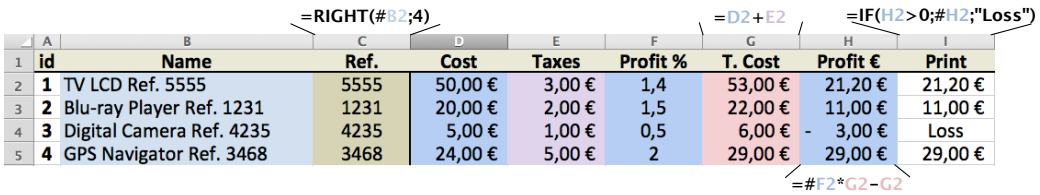
Notwithstanding, formulas in a traditional spreadsheet are inherently unidirectional: when a user edits the value of a cell, the spreadsheet recalculates all the corresponding formulas to display the updated results, but the values of the target cells are not editable by users. For instance, a cell $C = A + B$ computes the value of C from the values of A and B, but the user is not able to compute the values of A and/or B by editing C. For many cases, this restriction is unnecessary, in the sense that there are natural (but not necessarily unique) choices to reverse the computations, and limits the usefulness of the system.

That said, it is natural to imagine being able to evaluate spreadsheet formulas bidirectionally: users edit the values of the target cells of a formula, and input cells are automatically updated accordingly. Such a feature may be useful when two formats are interchangeable, to reverse arithmetic formulas, to fix errors, or to modify information directly in a summary table. Since spreadsheet formulas can refer to other cells, this also prevents users from having to trace multiple steps backwards, a tedious task for long calculation chains [4]. More intricate examples involve "what-if" scenarios where it is easier to change the output and inspect how it possibly affects the input cells, instead of trying to modify the inputs on a trial-and-error basis until obtaining the desired outputs—which is probably a "technique" attempted by many non-professional users.

Consider the spreadsheet for the forecast of profits depicted in Fig. 1. Each row represents a *product* whose first column defines its identifier, the second and third its name and reference (extracted using the RIGHT function from the name), and the next three its production cost, taxes cost and profit margin. The last column presents a summary report of the total expected profit (calculated in column H), and is processed with an IF statement: profits are simply presented as numerical values, and losses are alerted with the string `"Loss"`. Thus, a modification to a cost will trigger the recalculation of the resulting profit, but the opposite is not possible: one can not simply modify the profit and trigger a recalculation of the costs. Other possible examples of bidirectional spreadsheets include calculation of tax deductions, when the taxpayer wants to estimate the expenses that he needs to declare to attain a maximum tax deduction, or bet winning calculations, when the gambler wants to decide where or how much to bet by tuning the resulting gains.

Being able to calculate spreadsheet formulas backwards without having to redesign the existing calculation flow is not a new idea, and a couple of systems in the past have proposed to leverage traditional spreadsheets with new spreadsheet-like interfaces that allow calculations to be executed in multiple directions [5]–[8]. The novelty of these systems often comes from changing the spreadsheet model into the world of constraint programming where users specify constraints that express relationships between cells, and the system automatically updates the values of spreadsheet cells as users enter or delete values from related cells. However, this asks users to (again!) adapt into a new spreadsheet interface with a somewhat different programming experience. In particular, they need to learn a new constraint language and become aware of the different (and in some cases unclear) implications of editing cells as precluded by the new interface. This is a significant burden especially for non-proficient spreadsheet users, that are already familiar with the commonplace spreadsheet formula language.

Starting from first principles, any spreadsheet extension should be 1) *intuitive*, by providing a seamless integration with



	A	B	C	D	E	F	G	H	I
1	**id**	**Name**	**Ref.**	**Cost**	**Taxes**	**Profit %**	**T. Cost**	**Profit €**	**Print**
2	**1**	TV LCD Ref. 5555	5555	50,00 €	3,00 €	1,4	53,00 €	21,20 €	21,20 €
3	**2**	Blu-ray Player Ref. 1231	1231	20,00 €	2,00 €	1,5	22,00 €	11,00 €	11,00 €
4	**3**	Digital Camera Ref. 4235	4235	5,00 €	1,00 €	0,5	6,00 €	- 3,00 €	Loss
5	**4**	GPS Navigator Ref. 3468	3468	24,00 €	5,00 €	2	29,00 €	29,00 €	29,00 €

Fig. 1. Example spreadsheet and update propagation.

traditional development processes, not requiring to learn new languages or new spreadsheet-specific features, 2) *conservative*, by not affecting the usual behavior of existing spreadsheets, and 3) *transparent*, by presenting the new features using terminology that users are already familiar with. This is to say that the ability to evaluate formulas backwards should not compromise the nature nor the evaluation of existing spreadsheet formulas, and it should be possible to reason about the effects of backward evaluation in terms of the existing spreadsheet model.

In this paper, we advocate this lightweight approach: we propose a technique to interpret spreadsheet formulas as lenses, i.e., BXs that can be *intuitively* run forwards to evaluate a formula and backwards to propagate updates on formula cells to a set of source cells selected by the user (from among the cells referenced in the dependency graph of the formula). To keep the design simple, our approach focuses on the (bidirectional) semantics of individual spreadsheet formulas and is fully integrated with Microsoft Excel. Our extension is *conservative* and only requires a minimal extension to the interface: users can explicitly mark cells that they wish to evaluate bidirectionally—this is essential to preserve the usability of the system and keep it predictable to the user. It is also *transparent*, by allowing the users to inspect (and eventually control) the synthesized backward transformations as spreadsheet formulas themselves.

Still, bidirectional spreadsheet formulas are not always desirable; for instance, in our forecast example we probably do not want taxes to be changeable, as they are institutionally fixed. They are much less trivial: a formula may refer to several cells or multiple times the same cell, refer to a cell that is itself a formula, inducing a chain of computations, or contain complex expressions, e.g. involving several conditionals. In general, formulas are not surjective nor injective and the corresponding backward transformation will be non-deterministic, meaning that the input values could be updated in various ways to produce the new output, requiring the system (or the user) to provide some sensible decision procedure. This is the usual scenario tackled by existing BX systems [1], [9]–[12], that offer mechanisms to bidirectionalize formulas that are typically not injective. The multi-disciplinary context of spreadsheets also makes them a unique scenario for applying and combining ideas from different BX domains. In fact, we believe that spreadsheets can become the "killer" application for BXs!

Section II introduces our BX framework for spreadsheets. Section III motivates the need for cell invariants, Section IV analyzes how these can be explored in a BX and Section V applies these concepts to some spreadsheet primitives. An implementation as a Microsoft Excel add-in is presented in Section VI. Section VII discusses related work and Section VIII finishes with conclusions and directions for future work.

II. Bidirectionalizing Spreadsheet Formulas

A. Bidirectional Spreadsheet Formulas

A *lens* [1] is a bidirectional transformation with two components: a forward transformation $\mathsf{get} : A \to B$ that computes a target of type B from a source of type A, and a backward transformation (or *putback*) $\mathsf{put} : B \times A \to A$ that takes an original source and an updated target and produces a new updated source. Since get may drop source information, the original source passed to put allows it restore data not present in the target. Thus, the framework of lenses is asymmetric, and works better for scenarios where B contains less information than A. A lens is *well-behaved* if the following properties hold.

$$\mathsf{get}\ (\mathsf{put}\ (b, a)) = b \qquad \textsc{PutGet}$$
$$\mathsf{put}\ ((\mathsf{get}\ a), a) = a \qquad \textsc{GetPut}$$

PutGet guarantees that user updates on the target are preserved by a round-trip, while GetPut guarantees that the system is stable for null updates. Various BX approaches encompass the design of combinatorial domain-specific lens languages whose combinators denote well-behaved pairs of get and put transformations, allowing users to build complex correct-by-construction programs by composition [1], [9], [11], [12].

In our approach, the subjects of bidirectionalization are ordinary spreadsheet formulas. Spreadsheets are naturally reactive, since an update on a cell is automatically propagated to all formula cells that depend on it. Bidirectional formulas should react in the same way, by automatically propagating updates on a target cell to input cells that it refers to. Such kind of BXs are said to work in an *online* setting [13] and typically allow a finer control of the update propagation by collecting information from the environment, in contrast to *offline* approaches that do not rely on system-specific information. In spreadsheets, this information may include constraints imposed by other formulas or cell bindings in the current environment. Unlike combinatorial BX languages we will process each formula as an independent BX, what will allow us to keep a simple design while handling the necessary environment information.

As a change in a cell may prompt the recalculation of a chain of multiple dependent formulas, the update of input cells prompted by backward propagations may trigger the backward and forward recalculation of other dependent formulas. In normal spreadsheets systems, formulas cannot build cycles to avoid infinite calculation loops–neither should the evaluation of bidirectional formulas. The required properties to guarantee that update propagation does not loop indefinitely and converges into a consistent state are exactly the round-tripping properties of lenses [1]: 1) if a cell is not updated then its formula never has to be recomputed and 2) a backward evaluation never requires the forward evaluation for the same formula.

In general, updates can be reflected back in more than one way to the input cells. In order to keep the system predictable, we adopt a *conservative updating* principle and ask users to explicitly indicate, for each formula, which cells can be updated, by marking them with the special symbol #. E.g., the bidirectional formula from column H in Fig. 1 only updates the value of profit margins (F). This ensures no modifications occur unless authorized by the user—formulas without # marks behave as ordinary unidirectional ones. In order to allow the user to understand the backward semantics (and eventually parameterize it), we follow a *white-box* approach, specifying all backward transformations as spreadsheet formulas themselves.

Taking these design decisions into consideration, a formula f on a cell B that depends on input cells $A_1, ..., A_n$ will be denoted as $f : A_1 \times ... \times A_n \rightarrow B$ (for readability, $A_1 \times .. \times A_n$ will often be abbreviated to A). For each #-marked cell an individual put formula is synthesized, whose behavior depends on the other # marks of the formula. To characterize the different scenarios, each put is indexed with extra marks: ■ denotes the cell for which the put is being defined; □ denotes another #-marked cell; and cell names denote a non-marked constant parameter. E.g., for a formula $C = \#A + \#B$ where both A and B are #-marked, both $\mathsf{put}_{\blacksquare+\square} : C \times (A \times B) \rightarrow A$ and $\mathsf{put}_{\square+\blacksquare} : C \times (A \times B) \rightarrow B$ are synthesized, to update the values on A and B when C is updated; for a formula $C = \#A + B$ only a $\mathsf{put}_{\blacksquare+B} : C \times (A \times B) \rightarrow A$ for cell A is required (which is not the same as $\mathsf{put}_{\blacksquare+\square}$). Cell *ranges* may represent arrays, denoted by $(A_1 : A_n)$, or matrices, represented by $(A_{1,1} : A_{m,n})$, with A_i denoting the array corresponding to column i. A # mark on a range is assumed to affect every cell in it, thus $\#(A_1 : A_n)$ (similar to $(\#A_1, \#A_2, \#(...), \#A_n)$) is distinguished from $(\#A_1 : \#A_n)$ (similar to $(\#A_1, A_2, ..., \#A_n)$). Since many functions, like SUM or MAX, allow ranges to be decomposed, this improves the control of the user over the propagation of updates.

B. Spreadsheet Primitive Function Examples

Modern spreadsheet systems support database-like functions (e.g. VLOOKUP), functions for the manipulation of strings (e.g. LEFT, LEN) or arithmetic operations (e.g. MAX, SUM), and logical operators (e.g. IF). Here we present first attempts to bidirectionalize some basic functions. Section III will show why these definitions would be too inflexible for practical situations.

Let us start with a simple addition with a single #-marked cell, like $B = \#A_1 + A_2$. Since this formula is injective, the corresponding putback has a single valid solution:

$$\mathsf{put}_{\blacksquare+A_2}\ (b, (a_1, a_2)) = b - a_2$$

If both references are #-marked as $B = \#A_1 +\# A_2$, b can be divided into A_1 and A_2 in any way as long as their addition is b. One option is to divide b by two:

$$\mathsf{put}_{\blacksquare+\square}\ (b, (a_1, a_2)) = \mathsf{put}_{\square+\blacksquare}\ (b, (a_1, a_2)) = b\ /\ 2$$

This can be generalized for the n-ary SUM operation.

A typical operation over a string is to compute its length using the LEN function. One reasonable putback formula retrieves as much as possible from the original string:

$$\mathsf{put}_{\mathsf{LEN}\ (\blacksquare)}\ (b, a) = \mathbf{if}\ b \leqslant \mathsf{LEN}\ (a)\ \mathbf{then}\ \mathsf{LEFT}\ (a, b)\ \mathbf{else}\ a\ \&\ \mathsf{REPEAT}\ (\texttt{"A"}, b - \mathsf{LEN}\ (a))$$

VLOOKUP is used to find information in large data tables, and its putback diverges greatly depending on the #-marked cells. When the table array is #-marked and the remaining arguments are constant ($D = \mathsf{VLOOKUP}\ (A, \#(B_{1,1} : B_{m,n}), C)$), we can simply update the cell retrieved by VLOOKUP:

$$\forall i, j.\ \mathsf{put}_{\mathsf{VLOOKUP}\ (A,(\square_{1,1}:\blacksquare_{i,j}:\square_{m,n}),C)}\ (d, (a, b, c)) = \mathbf{let}\ r = \mathsf{MATCH}\ (a, b_1, 0)\ \mathbf{in\ if}\ (i, j) \equiv (c, r)\ \mathbf{then}\ d\ \mathbf{else}\ b_{i,j}$$

For an IF conditional without # marks on the conditional test ($D = \mathsf{IF}\ (A, \#B, \#C)$), its putback can be defined by updating the input cell that was originally copied to the result:

$$\mathsf{put}_{\mathsf{IF}\ (A,\blacksquare,\square)}\ (d, (a, b, c)) = \mathbf{if}\ a\ \mathbf{then}\ d\ \mathbf{else}\ b$$
$$\mathsf{put}_{\mathsf{IF}\ (A,\square,\blacksquare)}\ (d, (a, b, c)) = \mathbf{if}\ a\ \mathbf{then}\ c\ \mathbf{else}\ d$$

C. Spreadsheet Formula Chaining

Until now, we have focused on the bidirectionalization of individual primitive functions. In spreadsheets, functions can be composed in two ways: either through *formula nesting* (by defining complex formulas as $f(g(A))$) or through *formula chaining* (by having $B = f(A)$, where A is itself a formula $A = g(B)$). For simplicity, we focus on the latter, since nested formulas can be decomposed into chains of formulas[1]. The backward evaluation of formula chains exploits the reactive nature of spreadsheets: updating an intermediate formula cell will trigger the backward evaluation of its own formula. Consider a concrete example, with cells $D = \#C + \#B$, $C = \mathsf{LEN}\ (\#A) = 5$, $B = 10$ and $A = \texttt{"hello"}$. An update $D \leftarrow 8$ is propagated back as $C \leftarrow \mathsf{put}_{\blacksquare+\square}\ (8, (10, 5)) = 4$ and $B \leftarrow \mathsf{put}_{\square+\blacksquare}\ (8, (10, 5)) = 4$. Since B is a value cell, the chain stops; C is a formula cell, so another update $A \leftarrow \mathsf{put}_{\mathsf{LEN}\ (\blacksquare)}\ (4, \texttt{"hello"}) = \texttt{"hell"}$ is triggered.

To be sound, formula chaining requires a careful analysis of the formula dependencies. First, formulas cannot be cyclic (what is not an issue since spreadsheet systems already ensure this restriction). Second, all #-marked references must eventually lead to an updatable cell (i.e., a #-marked value cell). To understand this restriction, imagine two cells $C = f(\#B)$ and $B = g(A)$. If C is updated, the system would propagate the update to B, but would not be able to propagate it to A as g is not a bidirectional formula. This is a premeditated design decision: we could implicitly allow g to be evaluated bidirectionally, but we prefer to make users explicitly aware of the implications of backward propagations. Third, a cell occurring multiple times in the dependency graph of a formula can never be #-marked. Updates must be performed locally, without inspecting other formulas—the same cell appearing more than once would require a global analysis of the spreadsheet. Consider formulas $C = \#A + \#B$ and $B = \#A$; to work properly, the putback at C would need to infer that the values propagated to A and B need to be the same[2].

A particular line of BX approaches considers the problem of supporting data duplication [13], [14], for which they relax the BX properties and develop sophisticated BX semantics.

[1] For instance, $B = f(g(A))$ can be rewritten to $B = f(X)$ and $X = g(A)$, with an auxiliary cell X.

[2] This would not be an issue for logical spreadsheet systems, where the relationship between A and B could be specified as a global constraint.

Our framework naturally allows duplication to a certain degree (for formulas with disjoint dependency graphs), even with the standard properties and the above restrictions. For example, consider $B = f(\#A)$ and $C = g(A)$; updating B assigns a new value to A through f, which in turn triggers a forward evaluation of g, restoring the consistency between A and C.

III. Invariants

Let us return to the forecast example from Fig. 1. It is easy to see that the output of the formula in column I is either a positive number or the string `"Loss"`. Thus, user updates in these bidirectional cells must be somehow restricted: a negative value has no source values that output it (breaking PutGet). Similarly, in the references column C calculated by a RIGHT function, any string with length higher than 4 is outside its range. Since spreadsheet formulas are not *surjective* in general[3], we need a way to check if an updated value is within the domain of a bidirectional formula. This problem is aggravated by chains of bidirectional formulas: the putback of the second formula must generate values within the range of the first one. IF statements bring additional complexity: in an update from `"Loss"` to a positive number (column I), the putback of IF should support the change of branch and update the respective value in column H; the naive semantics from Section II renders this impossible.

Now imagine that the user inserted a constraint in column F stating that the profit margin can never exceed 200% (using, for instance, the 'Data Validation' feature of Excel). The domain of allowed values in the I cells must be coherently restricted, to forbid the insertion of profit values that will exceed the 200% margin. For instance, any value higher than 53 in the column I of product 1 would result in an invalid update. These two scenarios (non-surjective formulas and user-defined constraints) motivate the introduction of *invariants* in our BX framework, to validate updated target values and guide the generation of source values during backward propagation. We will assume user-defined constraints to be applied only at source cells[4]. These must be propagated through chains of formulas, as they also affect the domain of valid output values.

Formally, invariants (represented by upper-case greek letters $\Phi, \Psi \in \mathcal{P}(\phi)$) consist of sets of clauses (represented by lower-case greek letters $\varphi, \psi \in \phi$) that denote sets of values. The set of all values allowed by an invariant is defined as the union of all values in its clauses. For a cell A, Φ_A will denote an invariant Φ restricting the values of A, and similarly for φ_A. We support clauses over strings, reals, integers or booleans, according to the following abstract syntax:

$$
\begin{array}{lll}
\phi & \in & \mathsf{Num} \mid \mathsf{Int} \mid \mathsf{Text} \mid \mathsf{Bool} \\
\mathsf{Num} & \in & \langle \mathbb{R}..\mathbb{R} \rangle \mid \langle \mathbb{R}..\mathbb{R}\langle \mid \rangle \mathbb{R}..\mathbb{R} \rangle \mid \rangle..\mathbb{R} \rangle \mid \langle \mathbb{R}..\langle \mid \mathsf{Univ}_{\mathbb{R}} \\
\mathsf{Int} & \in & [\mathbb{Z}..\mathbb{Z}] \mid [\mathbb{Z}..[\mid]..\mathbb{Z}] \mid \mathsf{Univ}_{\mathbb{Z}} \\
\mathsf{Text} & \in & \Sigma^* \mid \mathsf{len}_{\mathsf{Int}} \mid \mathsf{Univ}_{\Sigma^*} \\
\mathsf{Bool} & \in & \mathsf{True} \mid \mathsf{False} \mid \mathsf{Univ}_{\mathsf{Bool}}
\end{array}
$$

Here, clauses Σ^* denote constant strings of arbitrary length, and clauses Univ_ϕ accepts any value in ϕ. For simplicity, we will often write constants $[x..x]$ or $\langle x..x \rangle$ as x, and integer intervals $[x..y-1]$ and $[x+1..y]$ as open intervals $[x..y[$ and $]x..y]$, respectively. Note that for $x, y \in \mathbb{Z}$, interval $\langle x..y \rangle$ is continuous while $[x..y]$ is discrete. Most of these clauses are inspired by the data constraint features of Excel and the constraint languages of logical spreadsheet systems [5], [6], that usually support numerical intervals. The notable exception is the $\mathsf{len}_{\mathsf{Int}}$ clause that denotes the set of strings whose length belongs to the integer parameter. E.g., $\mathsf{len}_{[0..10]}$ represents all strings whose length is between 0 and 10. To be manageable, invariants are processed in a normalized form, such that their comprising clauses are disjoint. Our invariants can deal with numerical formulas whose range is representable by a finite set of intervals. E.g., the square $A\hat{}2$ over integers is not definable, as it would require an infinite union of singleton sets (though the square of reals works fine). For strings, it supports formulas which are oblivious to the string's content (except constants); we cannot give precise invariants for functions like UPPER. Even so, this simple language of invariants is already sufficient to solve interesting BX examples.

Some operations are required to test and manipulate these invariants. The membership test $x : \Phi$ boils down to testing if x belongs to any of the clauses $\varphi \in \Phi$; union $\cup$ and intersection $\cap$ of invariants produce themselves normalized invariants, e.g.:

$$
\begin{array}{l}
\{[x..y]\} \cup \{[a..b]\} = \textbf{if } a \leqslant y \ \vee \ x \leqslant b \\
\quad \textbf{then } \{[\mathsf{min}\ (x,a)..\mathsf{max}\ (y,b)]\} \textbf{ else } \{[x..y],[a..b]\} \\
\{[x..y]\} \cap \{[a..b]\} = \textbf{if } a \leqslant y \ \vee \ x \leqslant b \\
\quad \textbf{then } \{[\mathsf{max}\ (x,a)..\mathsf{min}\ (y,b)]\} \textbf{ else } \{\}
\end{array}
$$

We will also require the difference ($-$) operation on invariants; however, for particular cases over strings the resulting invariant may not be expressible in a normalized form (e.g. $\mathsf{len}_\varphi - y$, for $\mathsf{LEN}\ (y) \cap \varphi \neq \emptyset$). The $\mathsf{sel} : \mathcal{P}(\phi) \times A \to A$ operation receives an invariant and an original value (not necessarily satisfying the invariant), and generates a repaired value that satisfies the invariant; already valid values shall not be repaired:

$$\forall a.\ a : \Phi \Rightarrow \mathsf{sel}\ (\Phi, a) = a \qquad \textsf{sel-Stable}$$

At this point, readers may see sel as an abstract placeholder denoting a semi-arbitrary choice in the backward propagation of a formula[5]; in practice, it will be possible to control this choice for particular situations (Section VI).

IV. Synthesis of Putback Formulas

A. Overview

In order to evaluate a function $f : A \to B$ backwards, the system needs to be able to, given an updated target b, find one possible source value a that is consistent with b and satisfies the existing source invariant Φ_A, i.e., find a value that satisfies the invariant $f^{-1}\ b \cap \Phi_A$. Since the possibly non-functional inverse f^{-1} of a function f is not easily computable in general, we instead define, for each particular bidirectionalizable function in our add-in, the target range of f over a normalized source invariant; such target invariant declares the set of valid updates on the target of a formula. Furthermore, in order to propagate target updates, the putback formula needs to establish a correspondence between valid target values and corresponding source values. Thus we compute, for each formula and source invariant, a target invariant Φ_B and a traceability link between

[3] Spreadsheets are not typed, so there are technically no surjective functions.

[4] The evaluation of formulas in systems like Excel actually ignores invariants, outputting any value produced by a formula regardless of the constraint.

[5] It plays a similar role to the missing Ω placeholder of [1] and the *create* function of [12], but generating default values within particular invariants.

(values in) it and the source invariant. Using this information, we define a backward transformation that is guaranteed to succeed for source and target values within the invariants.

In this section, we present the general algorithm for synthesizing putback formulas for arbitrary spreadsheet primitive functions. Examples illustrating its application to particular primitives are presented in Section V. Our algorithm synthesizes a putback formula locally, i.e., for each formula cell, and statically, for the constraints in a given state of the spreadsheet. To avoid an expensive global analysis of the spreadsheet, our invariants only express predicates over single cells, by freezing the values of other cells as constants. E.g., a constraint $A \leqslant B$, where $A = 5$ and $B = 10$ is decomposed into two invariants $\Phi_A =]..10]$ and $\Phi_B = [5..[$. This breaks down the complexity of the system and allows each bidirectional formula to be handled independently. The caveat is that modifying a bidirectional formula, user-defined constraint affecting input cells, or cell referenced by source invariants, requires the synthesis of a new put consistent with the new state of the spreadsheet. Since each #-marked input cell is assigned an independent put, constraints relating multiple input cells (like $A \leqslant B$) impose a sequential order on updates: e.g., updating $A \leftarrow a$ first, refreshing the invariant on B to $\Phi_B = [a..[$, and then updating B taking into consideration the updated invariant (or vice-versa).

B. Synthesis Procedure

For a function $f : A \to B$, our general putback synthesis algorithm can be defined according to the following steps[6]:

1) Calculate the invariant $\Phi_A \in \mathcal{P}(\phi_A)$ over A, taking into consideration the domain of f (δf) and any pre-existing constraint over A not imposed by f;
2) Calculate the target range Φ_B of f over the source invariant Φ_A and the traceability link $\overline{f\ \Phi_A} \subseteq \mathcal{P}(\phi_B) \times \mathcal{P}(\phi_A)$ between source and target invariants;
3) For every marked input cell $\#A_i$ and for each trace link $(\Psi_B, \Psi_A) \in \overline{f\ \Phi_A}$, synthesize a putback component $\overleftarrow{f\ (..., \blacksquare_i, ...)}\ \Psi_A : B \times A \to A_i$ that, given an updated value $b : \Psi_B$ and an original value $a : \Phi_A$, produces a consistent $a_{i'} : \Psi_{A_i}$;
4) For every marked input cell $\#A_i$, synthesize the putback $\mathtt{put}_{f\ (..., \blacksquare_i, ...)} : B \times A \to A_i$ that for an updated value $b : \Psi_B$ and an original value $a : \Phi_A$, given the trace link $(\Psi_B, \Psi_A) \in \overline{f\ \Phi_A}$, calls the appropriate $\overleftarrow{f\ (..., \blacksquare_i, ...)}\ \Psi_A$.

For every function $f : A \to B$, we require a *domain* operation $\delta f \in \mathcal{P}(\phi_A)$, that computes the domain invariant of f. Furthermore, we require an operation $\rho f : \phi_A \to \mathcal{P}(\phi_B)$, that computes the *range* of a source invariant clause, that is:

$$\forall b.\ b : \rho f\ \varphi \equiv (\exists a.\ a : \varphi \Rightarrow f\ a = b).$$

Given an invariant Φ_A, the *range* Φ_B of f over Φ_A will be denoted by $\rho f\ \Phi_A \in \mathcal{P}(\phi_B)$ (overloading ρ) and defined as the union of all $\rho f\ \varphi$ such that $\varphi \in \Phi_A$.

We then calculate the traceability $\overline{f\ \Phi_A} \subseteq \mathcal{P}(\phi_B) \times \mathcal{P}(\phi_A)$ between source and target invariants. The left projection of $\overline{f\ \Phi_A}$ must form a partition of the range Φ_B. Also, for every pair $(\Psi_B, \Psi_A) \in \overline{f\ \Phi_A}$ and value $b : \Psi_B$, there must exist at least one consistent source a in every clause $\psi_A \in \Psi_A$:

$$\forall b : \Psi_B.\ \forall \psi_A \in \Psi_A.\ \exists a : \psi_A.\ f\ a = b \tag{1}$$

For instance, consider a function $f(A) = A\hat{}2$, with a pre-existing domain invariant $\Phi_A = \{\langle 0..10\rangle, \langle -6..-5\rangle\}$. Applying the range definition yields $\rho(A\hat{}2)\ \Phi_A$:

$$\begin{array}{ll} \langle 0..10\rangle & \mapsto \langle 0..100\rangle \\ \langle -6..-5\rangle & \mapsto \langle 25..36\rangle \end{array}$$

With range invariant $\Phi_B = \{\langle 0..100\rangle\}$. The ϕ_B clauses overlap, as values $\langle 25..36\rangle$ are produced by both $\langle 0..10\rangle$ and $\langle -6..-5\rangle$. Since the ϕ_B component of the normalized traceability must form a partition, these must be split, and thus $\overline{A\hat{}2\ \Phi_A}$ becomes:

$$\begin{array}{ll} \{\langle 0..25\langle, \rangle 36..100\rangle\} & \mapsto \{\langle 0..10\rangle\} \\ \{\langle 25..36\rangle\} & \mapsto \{\langle -6..-5\rangle, \langle 0..10\rangle\} \end{array}$$

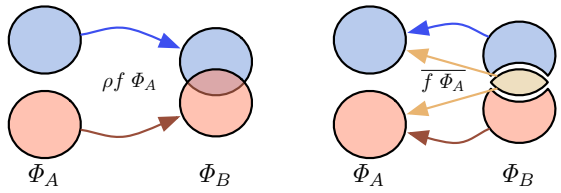


Fig. 2. Representation of the traceability link $\overline{f\ \Phi_A}$.

Fig. 2 depicts this process graphically, by comparing $\rho f\ \Phi_A$ with the produced $\overline{f\ \Phi_A}$. By keeping this traceability in a normalized form, the source invariant into which the updated a' source must fall for each target b can be easily found.

For every marked input cell $\#A_i$ of f and each trace pair (Ψ_B, Ψ_A) we require an operation $\overleftarrow{f\ (..., \blacksquare_i, ...)}\ \Psi_A$ that, for an updated target $b : \Psi_B$ and an original source a, produces a consistent update $a_{i'} : \Psi_{A_i}$ for the cell A_i. These operations shall be statically synthesized, for a particular invariant Φ_A and function f, and are the components that actually describe the behavior of each putback formula. The invariant parameter Ψ_A from the traceability is assumed to be normalized, and from (1) we know that $\overleftarrow{f\ (..., \blacksquare_i, ...)}\ \Psi_A$ may select the new source a' from *any* of the invariant clauses $\psi_A \in \Psi_A$. The top-level putback simply assembles these components together: based on traceabilities (Ψ_B, Ψ_A), $\mathtt{put}_{f\ (..., \blacksquare_i, ...)}$ tests to which Ψ_B invariant b belongs and applies the corresponding $\overleftarrow{f\ (..., \blacksquare_i, ...)}\ \Psi_A$. From the properties of the traceabilities and putback components, put is correct-by-construction.

V. Examples of Instantiations

Each spreadsheet primitive function has a particular backward component synthesis procedure. In this section we present the instantiation of some of the supported functions.

A. LEN Function

The synthesis procedure for the LEN function follows the same idea of the unbounded version presented in Section II-B, but now taking into consideration the respective cell invariants. If the target length is decreased, it will "try" to trim the original string; if it increases, it will "try" to use the original string as a prefix; if the preserving part of the original source does not

[6] We write $\phi_A = \phi_{A_1} \times ... \times \phi_{A_n}$ and $\varphi_A = (\varphi_{A_1}, ..., \varphi_{A_n})$.

satisfy the source invariant, it will select a consistent string using sel. The LEN function is total for any string ($\delta\mathsf{LEN} = \{\mathsf{Univ}_{\Sigma^*}\}$). Its range for a given clause is defined as:

$$\rho\mathsf{LEN}\ (x \in \Sigma^*) = \{\mathsf{LEN}\ (x)\}$$
$$\rho\mathsf{LEN}\ \mathsf{len}_x = \{x\}$$
$$\rho\mathsf{LEN}\ \mathsf{Univ}_{\Sigma^*} = \{\mathsf{Univ}_{\mathbb{Z}}\}$$

The putback components are synthesized by a procedure that tries to produce optimal source updates taking into consideration the invariants. For instance, if the update b is less or equal than $\mathsf{LEN}\ (a)$, and clauses Univ_{Σ^*} or len_x, with $b : x$, are available in Ψ_A, it simply trims the original string; next, it searches for a prefix of the original string in the available string constants; if none is found, it calls sel to generate an arbitrary consistent string. Since the traceability is normalized according to (1), additional tests are unnecessary. If the length increases instead, the behavior is similar, but searching for an adequate extension. The produced putback also preserves GETPUT.

Consider $\Phi_A = \{\texttt{"abc"}, \texttt{"xyz"}, \mathsf{len}_{[4..10]}\}$ (note that $\delta\mathsf{LEN} \cap \Phi_A = \Phi_A$), resulting in traceability $\overline{\mathsf{LEN}\ (A)\ \Phi_A}$

$$3 \mapsto \{\texttt{"abc"}, \texttt{"xyz"}\}$$
$$[4..10] \mapsto \{\mathsf{len}_{[4..10]}\}$$

with range invariant $\rho(\mathsf{LEN}\ (A))\ \Phi_A = \{3, [4..10]\}$. The synthesized $\mathsf{put}_{\mathsf{LEN}\ (\blacksquare)}$ for the invariant Φ_A is then:

$$\begin{array}{l}
\mathsf{put}_{\mathsf{LEN}\ (\blacksquare)}\ (b, a) = \\
\quad \textbf{if}\ b : \{3\}\ \textbf{then} \\
\quad\quad \textbf{if}\ b \leqslant \mathsf{LEN}\ (a)\ \textbf{then} \\
\quad\quad\quad \textbf{if}\ \mathsf{LEFT}\ (b, a) = \texttt{"abc"}\ \textbf{then}\ \texttt{"abc"} \\
\quad\quad\quad \textbf{else if}\ \mathsf{LEFT}\ (b, a) = \texttt{"xyz"}\ \textbf{then}\ \texttt{"xyz"} \\
\quad\quad\quad \textbf{else}\ \mathsf{sel}\ (\{\texttt{"abc"}, \texttt{"xyz"}\}, \mathsf{LEFT}\ (b, a)) \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{if}\ \mathsf{LEFT}\ (b, \texttt{"abc"}) = a\ \textbf{then}\ \texttt{"abc"} \\
\quad\quad\quad \textbf{else if}\ \mathsf{LEFT}\ (b, \texttt{"xyz"}) = a\ \textbf{then}\ \texttt{"xyz"} \\
\quad\quad\quad \textbf{else}\ \mathsf{sel}\ (\{\texttt{"abc"}, \texttt{"xyz"}\}, a) \\
\quad \textbf{if}\ b : \{[4..10]\}\ \textbf{then} \\
\quad\quad \textbf{if}\ b \leqslant \mathsf{LEN}\ (a)\ \textbf{then}\ \mathsf{LEFT}\ (b, a) \\
\quad\quad \textbf{else}\ a\ \&\ \mathsf{sel}\ (\{\mathsf{len}_{b-\mathsf{LEN}\ a}\}, a)
\end{array}$$

If the updated b is 3, it searches the constants for the closest string; otherwise the $\mathsf{len}_{[4..10]}$ clause allows it to freely generate the closest solution. This putback is automatically synthesized and could eventually be simplified: if $b : \{3\}$ is true, $b \leqslant \mathsf{LEN}\ (a)$ always holds. Section VI will show how this intermediary notation is translated to the spreadsheet formula language, by instantiating the invariant tests and sel operations.

B. IF *Statement*

An interesting spreadsheet function is the $\mathsf{IF}\ (C, \#A, \#B)$ logical statement, that affects the flow of update propagation. Without nested formulas, its putback simply needs to decide whether to propagate the update to A or B depending on the condition and the invariants. To make it manageable, predicate C (without # marks) is interpreted as a normalized invariant (Ψ_A, Ψ_B, K), with Ψ_A and Ψ_B invariants over A and B and K a constant predicate. Ψ_A and Ψ_B may contain references to each other, e.g., $A \geqslant B$ is interpreted as $([B..[,]..A], \mathsf{True})$. In comparison to the put from Section II, this version is able to change the selected branch; it also allows duplicated #-marked cells in the branches, like $\mathsf{IF}\ (C, \#A, \#A)$, as the update will only be propagated through one them. Its range must consider the original A and B values: if $\neg(b : \Psi_B)$, updates on A are not acceptable, as they will never render $b : \Psi_B$; if $\neg(a : \Psi_A)$, any B value is valid as it will never switch the branch (omitting K for the sake of simplicity):

$$\begin{array}{l}
\rho\mathsf{IF}\ ((\Psi_A, \Psi_B), \Phi_A, \Phi_B) = \\
\quad (\textbf{if}\ (b : \Psi_B)\ \textbf{then}\ (\Phi_A \cap \Psi_A)\ \textbf{else}\ \{\}) \cup \\
\quad (\textbf{if}\ (a : \Psi_A)\ \textbf{then}\ (\Phi_B - \Psi_B)\ \textbf{else}\ \Phi_B)
\end{array}$$

As a general logical combinator that just refers to other cells, we can provide a universal static definition for its putback formula instead of synthesizing it for specific invariants.

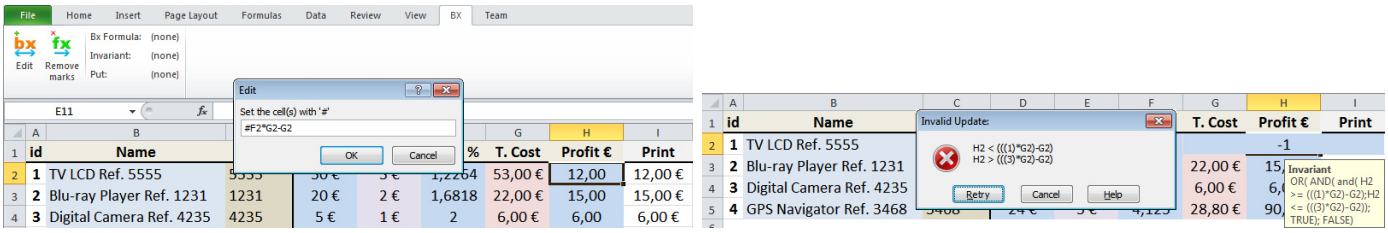
$$\begin{array}{l}
\mathsf{put}_{\mathsf{IF}\ (\Psi, \blacksquare, \square)}\ (c, (a, b)) = \\
\quad \textbf{if}\ c : \Phi_A\ \wedge\ \neg(c : \Phi_B)\ \textbf{then}\ c \\
\quad \textbf{if}\ \neg(c : \Phi_A)\ \wedge\ c : \Phi_B\ \textbf{then}\ a \\
\quad \textbf{if}\ (c, b) : \Psi\ \wedge\ (a, b) : \Psi\ \textbf{then}\ c \\
\quad \textbf{if}\ \neg((a, c) : \Psi)\ \wedge\ \neg((a, b) : \Psi)\ \textbf{then}\ a \\
\quad \textbf{if}\ (c, b)\ \textbf{in}\ \Psi\ \textbf{then}\ c\ \textbf{else}\ a
\end{array}$$

Here, $\Psi = (\Psi_A, \Psi_B)$. Only one of the branches is updated, so the put in each cell either returns the original source or the updated target. Since c is assumed to have passed the target invariant, it must be acceptable in one of the branches. The first two cases are straight-forward: if $c : \Phi_A$ but not $c : \Phi_B$, c must be propagated to A, and vice-versa. The next cases regard situations when $c : \Phi_A \cup \Phi_B$. The put tries to avoid switching the selected branch: if initially $(a, b) : \Psi$, and it remains so if c is propagated to A, A is updated; if instead $\neg(a, b) : \Psi$, and propagating c to B still evaluates Ψ to false, B is updated; the last cases occur when the selected branch must change. The definition of $\mathsf{put}_{\mathsf{IF}\ ((\Psi_A, \Psi_B), \square, \blacksquare)}$ is symmetric.

Although this putback works well in many cases, it can be made even more flexible by allowing both sides to be updated simultaneously: the cell opposite to the one receiving the update can be repaired in order to align C with the chosen branch. Although it would improve the updatability of the system, this invariant would not be representable in our language (as it would involve existential quantifications). To support it, we would need to relax the assumption that the invariant denotes the exact values for which backward propagation succeeds, and consider a weaker invariant that underapproximates it.

VI. IMPLEMENTATION

This section describes the prototype implementation of our BX system as an Excel add-in, depicted in Fig. 3. New functionalities are managed by a new tab named 'BX' in the Excel ribbon. To make a formula bidirectional (or modify an existing one), the user selects a formula and hits the 'Edit' button; a dialog box allows him to edit the formula with our special # symbol to mark updatable cells, as in Fig. 3a. The system checks if the new bidirectional formula satisfies our restrictions from Section II (no duplicated #-marked cells and all #-paths lead to value cells), and reports a faulty # mark if a violation is found; it also verifies if the user-defined constraints in input cells are expressible in our invariant language. Next, the system calculates the invariant on the target cell and synthesizes the respective putback—they are both shown to the user in the ribbon as spreadsheet formulas themselves. The 'Remove Marks' button converts a bidirectional formula into a unidirectional one by removing all # marks.



(a) Formula bidirectionalization.

(b) Invalid update.

Fig. 3. Screenshots of the BX add-in.

To trigger the backward evaluation of a bidirectional formula, the user simply writes the new value in the target cell, as he normally would. The system intercepts the event and, instead of overwriting the existing formula, propagates the update to the input cells. Fig. 3b depicts an invalid update: the user inserts a value in cell *H2* that violates the target invariant imposed by the formula (the profit can not be negative). Nested formulas are encoded in the implementation by internally storing virtual auxiliary cells, uniquely identified by their position inside the formula cell. Invariants are enforced by relying on Excel's 'Data Validation' feature, that allows the declaration of constraints over a cell that are enforced on each user update. We show invariants to the user by converting them into the disjunctive normal form; e.g., $\Phi_A = \{[0..10], 20\}$ becomes the formula $0 \leqslant A \leqslant 20 \vee A = 20$. In our current prototype, the $\mathsf{sel}(\Phi_A, a)$ operation is implemented by automatically repairing a into a value satisfying Φ_A. For numerical values, the Excel solver is used, and for strings a simple generation algorithm was implemented. In the next version, we plan to allow users to control this selection in the following way: by selecting a bidirectional formula, the system presents its put in the ribbon, together with a table 'invariant'/'original value'/'selected value', so that he can tweak the selected value (initially generated by default) considering the original value and the invariant to be satisfied.

VII. Related Work

A. Bidirectional Transformations

BX languages of interest for spreadsheet programming range from functional programming [1], [9] to arithmetic [10], string processing [12] or databases [11]. Our framework reuses many lessons learned in these languages for the synthesis of puts for particular formulas. But unlike them it is not combinatorial, in the sense that we only bidirectionalize formulas and the interaction between different cells is handled by the spreadsheet environment. Also, these languages often denote total and surjective transformations, by defining the domains of each program in standard type systems. Since spreadsheets, as a lightweight programming language, lack a type system, we considered precise invariants to describe the "types" for which formulas are total and surjective. We have previously studied the impact of introducing invariants in BXs [15].

The interactive bidirectional XML editor proposed in [13], like spreadsheets, reacts immediately to one operation at a time. To ensure that after each update the editor converges into a consistent state, transformations obey one-and-a-half round-tripping laws. To the best of our knowledge, existing work on the application of BX techniques to spreadsheets has not yet considered the bidirectionalization of spreadsheet formulas. In [16], we developed an OpenOffice plugin that tackles the bidirectional synchronization of spreadsheet models (modeling their business logic) and conforming instances.

B. Program Synthesis

Previous work has been done on the synthesis by example of spreadsheet transformations [17]. The process consists of defining languages, not necessarily in spreadsheet notation, for particular domains (strings, numbers and table layouts), for which a synthesizer produces transformations from input-output examples. The work of [18] proposes a technique for the synthesis of functions from decision procedures over a set of constraints, supporting integer arithmetic and collections. Our approach for addition follows their ideas, by representing constraints in disjunctive normal form, applying one-point rules to eliminate variables from the conjunctions, and then finding witnesses from one of them. A technique for the inversion of imperative programs through synthesis is proposed in [19]. While related to BX, pure inverse programs lack the ability to synchronize the original source with an updated view.

C. Spreadsheets

Significant work has been done in extending spreadsheets with logical programming capabilities [5], [8], culminating in a Workshop on Logical Spreadsheets (WOLS 2005). These approaches introduce Prolog-like operations to spreadsheet systems that could to some extent provide bidirectional behavior. Others have focused on extending spreadsheets with constraint-solving functionalities [6], [7], [20]. These can be applied to solve formulas backwards by specifying a set of constraint on the spreadsheet and a set of target cells to solve. However, at the user-interface level, solving is more tailored for search and optimization problems described through a set of global constraints over the spreadsheet, the spreadsheet acting as a mere interface to the solver. Typically, for every backward computation, the user must set up a new constraint-solving problem using a specialized interface. In both these extended systems, the user is expected to reason about bidirectionalization in a new language and/or interface—not using plain spreadsheet formulas. Our "solver" is not explicit and upfront but a backend: constraints are not first class entities that the user manipulates, but integrated seamlessly into the bidirectionalization approach.

Constraint-solvers typically support arithmetic constraints, but not high-level combinators like conditionals, table lookups or string manipulation, at which BXs excel. Nonetheless, for numerical functions our putbacks draw some similarities with solving. For instance, TK Solver [7] is able to solve

constraints with several target cells, restricted with additional cell constraints, as long as there is an initial "guess" to start the solver—in a BX, that role could be played by the original source values. However, it would defeat our white-box design principle, that we believe makes the bidirectionalization accountable and more easily trusted by users: in constraint-solving such "putback" is computed at runtime and in ways completely opaque to users. Still, our sel placeholders over numeric values currently use solving to generate valid defaults at synthesis time. We believe that our technique could benefit from localized solving procedures by postponing the concretization of sel values to runtime—once the overall sketch of the putback is synthesized, the user could either provide specific values for each sel placeholder, or call a solver to compute them.

GoalDebug [21] allows users to fix incorrect formulas by defining (numeric) constraints over their outputs: if the constraints are broken, the system proposes changes on the *formula* in order to restore consistency. These changes may involve modifying the formula or one of its parameters, being propagated until value cells are reached. While at first sight this technique resembles our own, they are fundamentally different: bidirectional transformations are meant to propagate updates between data domains, leaving formulas unchanged.

VIII. Conclusion and Future Work

In this paper, we have proposed a technique for the bidirectionalization of spreadsheet formulas, with a particular emphasis on the seamless integration with the standard spreadsheet development cycle. We have implemented our system as an add-in for Microsoft Excel. Owing to the fact that spreadsheets are not typed, we have also demonstrated how introducing invariants is essential to derive sufficiently flexible putback formulas, to provide users with errors for non-translatable modifications, and to ensure an acceptable level of predictability for translatable ones. Our approach was designed to obey three basic principles: to be 1) *intuitive*, in that users write ordinary formulas that can be run backwards just by editing their result cells; 2) *conservative*, as it preserves the forward behavior of formulas and backward evaluation only affects cells that are explicitly marked by users; and 3) *transparent*, in that backward transformations and invariants are presented to users as ordinary spreadsheet formulas themselves. Assessing if these goals are effective in practice requires a future empirical study.

The current implementation is only meant as a first prototype, and many extensions could and should be applied to make it a fully usable and effective BX environment. In the near future, we plan to improve the supported parameterization of sel operations appearing in putbacks, generalize our invariant language to capture more constraints (for example by admitting arbitrary regular expressions over strings), and provide better error reporting (for instance by suggesting to the user which cells in a chain deemed a particular update invalid). An advantage of our approach is that we work with the native spreadsheet formula language, with the caveat that we need to manually derive necessary information for each primitive function that we want to support. This is in a way similar to the Excel solver [20], that requires a previous analysis of each spreadsheet function that can be used in solved formulas. Although we have already managed to gather a considerable number of common primitives (like +, SUM, MAX or ^2 over numbers, RIGHT, LEFT or LEN over strings, VLOOKUP or FIND over tables, and IF statements), we plan to keep expanding the catalog of supported spreadsheet functions.

Acknowledgment

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by national funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FATBIT with reference FCOMP-01-0124-FEDER-020532. The first author is also sponsored by FCT grant SFRH/BD/69585/2010.

References

[1] J. N. Foster, M. Greenwald, J. Moore, B. 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, p. 17, 2007.

[2] D. Lutterkort, "Augeas: A configuration API," in *Linux Symposium*, July 2008, pp. 47–56, available from http://augeas.net/.

[3] B. Nardi, *A small matter of programming: perspectives on end user computing*. Cambridge, MA: The MIT Press, 1993.

[4] B. Nardi and J. Miller, "The spreadsheet interface: A basis for end user programming," in *INTERACT 1990*. North-Holland, 1990, pp. 977–983.

[5] M. Kassoff, L.-M. Zen, A. Garg, and M. Genesereth, "PrediCalc: a logical spreadsheet management system," in *VLDB 2005*. VLDB Endowment, 2005, pp. 1247–1250.

[6] Y. Adachi, "Intellisheet: a spreadsheet system expanded by including constraint," in *HCC 2001*. IEEE, 2001, pp. 173–179.

[7] M. Konopasek and S. Jayaraman, *The TK! Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education*. Osborne/McGraw-Hill, 1984.

[8] I. Cervesato, *The Deductive Spreadsheet*, ser. Cognitive Technologies. Springer, 2013.

[9] H. Pacheco and A. Cunha, "Generic point-free lenses," in *MPC 2010*, ser. LNCS, vol. 6120. Springer, 2010, pp. 331–352.

[10] T. Yokoyama, H. Axelsen, and R. Glück, "Principles of a reversible programming language," in *CF 2008*. ACM, 2008, pp. 43–54.

[11] A. Bohannon, B. Pierce, and J. Vaughan, "Relational lenses: a language for updatable views," in *PODS 2006*. ACM, 2006, pp. 338–347.

[12] A. Bohannon, J. N. Foster, B. Pierce, A. Pilkiewicz, and A. Schmitt, "Boomerang: resourceful lenses for string data," in *POPL 2008*. ACM, 2008, pp. 407–419.

[13] Z. Hu, S.-C. Mu, and M. Takeichi, "A programmable editor for developing structured documents based on bidirectional transformations," *Higher-Order Symb. Comput.*, vol. 21, no. 1–2, pp. 89–118, 2008.

[14] D. Liu, Z. Hu, and M. Takeichi, "Bidirectional interpretation of XQuery," in *PEPM 2007*. ACM, 2007, pp. 21–30.

[15] N. Macedo, H. Pacheco, and A. Cunha, "Relations as executable specifications: Taming partiality and non-determinism using invariants," in *RAMiCS 2012*, ser. LNCS, vol. 7560. Springer, 2012, pp. 146–161.

[16] J. Cunha, J. Fernandes, J. Mendes, H. Pacheco, and J. Saraiva, "Bidirectional transformation of model-driven spreadsheets," in *ICMT 2012*, ser. LNCS, vol. 7307. Springer, 2012, pp. 105–120.

[17] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Commun. ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[18] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Complete functional synthesis," in *PLDI 2010*. ACM, 2010, pp. 316–329.

[19] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, "Path-based inductive synthesis for program inversion," in *PLDI 2011*. ACM, 2011, pp. 492–503.

[20] D. Fylstra, L. Lasdon, J. Watson, and A. Waren, "Design and Use of the Microsoft Excel Solver," *Interfaces*, vol. 28, no. 5, pp. 29–55, 1998.

[21] R. Abraham and M. Erwig, "GoalDebug: A spreadsheet debugger for end users," in *ICSE 2007*. IEEE, 2007, pp. 251–260.