Type-safe Evolution of Spreadsheets

Jácome Cunha
jacome,@di.uminho.pt
Universidade do Minho
Portugal

Joost Visser
j.visser@sig.eu
Software Improvement Group
The Netherlands

Tiago Alves
t.alves@sig.eu
Universidade do Minho
Portugal
Software Improvement Group
The Netherlands

João Saraiva
jas@di.uminho.pt
Universidade do Minho
Portugal

July 28, 2010

Abstract

Spreadsheets are notoriously error-prone. To help avoid the introduction of errors when changing spreadsheets, models that capture the structure and inter-dependencies of spreadsheets at a conceptual level have been proposed. Thus, spreadsheet evolution can be made safe within the confines of a model.

As in any other model/instance setting, evolution may not only require changes at the instance level but also at the model level. When model changes are required, the safety of instance evolution can not be guarded by the model alone.

Coupled transformation of models and instances are supported by the 2LT platform and have been applied for transformation of algebraic datatypes, XML schemas, and relational database models.

We have extended 2LT to spreadsheet evolution. We have designed an appropriate representation of spreadsheet models, including the fundamental notions of formulæ, references, and blocks of cells. For these models and their instances, we have designed coupled transformation rules that cover specific spreadsheet evolution steps, such as extraction of a block of cells into a separate sheet or insertion of columns in all occurrences of a repeated block of cells. Each model-level transformation rule is coupled with instance level migration rules from the source to the target model and vice versa.

These coupled rules can be composed to create compound transformations at the model level that induce compound transformations at the instance level. With this approach, spreadsheet evolution can be made safe, even when model changes are involved.

1 Introduction

Spreadsheets are widely used by non-professional programmers, the so-called end users, especially to develop business applications. Spreadsheet systems offer end users a high level of flexibility, making it easier for people to get started working with spreadsheets. This freedom, however, comes with a price: spreadsheets are error prone as shown by numerous studies which report that up to 90% of real-world spreadsheets contain errors [12, 21, 22, 23].

As programming systems, spreadsheets lack the support provided by modern programming languages/environments, like for example, higher-level abstractions, and powerful type and modular systems. As a result, they are prone to errors. In order to improve the productivity of end users, several techniques have been recently proposed which guide end users to safely/correctly edit spreadsheets, like, for example, the use of spreadsheet templates [2], ClassSheets [7, 11], and the inclusion of visual objects to provide editing assistance in spreadsheets [9]. All these approaches propose a form of end user model-driven software development: First, a spreadsheet business model is defined, from which then a customized spreadsheet application is generated that guarantees the consistency of the spreadsheet data with the underlying model.

Despite of its huge benefits, model-driven software development is sometimes difficult to realize in practice due to two main reasons: First, as some studies suggest, defining the business model of a spreadsheet can be a complex task for end users [1]. As a result, they are unable to follow this spreadsheet development discipline. Second, things get even more complex when the spreadsheet model needs to be updated due to new requirements of the business model. End users need not only to evolve the model, but also to migrate the spreadsheet data so that it remains consistent with the model. To address the first problem, in [7] we have proposed a technique to derive the spreadsheet's business model, represented as a ClassSheet model, from the spreadsheet data. In this paper we address the second problem, that is, the co-evolution of the spreadsheet business model and the spreadsheet data (*i.e.*, the instance of the model).

Co-evolution of models and instances are supported by the two-level coupled transformation (2LT) framework [4] and has been applied for transformation of algebraic datatypes, XML schemas, and relational database models [10].

In this paper we extend 2LT to spreadsheet evolution. Therefore, we present an appropriate representation of a spreadsheet model, based on the ClassSheet business model [11], including the fundamental notions of formulæ, references, and expandable blocks of cells. For this model and its instance, we design coupled transformation rules that cover specific spreadsheet evolution steps, such as extraction of a block of cells into a separate sheet or insertion of columns in all occurrences of a repeated block of cells. Each model-level transformation rule is coupled with instance level migration rules from the source to the target model and vice versa. Moreover, these coupled rules can be composed to create compound transformations at the model level that induce compound transformations at the instance level. With this approach, spreadsheet evolution can be made type-safe, also when model changes are involved.

The rest of this paper is organized as follows. In Section 2 we discuss spreadsheet refactoring as our motivating example. In Section 3 we briefly describe the 2LT framework and introduce our model to represent spreadsheets. Section 4 defines the rules to

<>	A	B	C	D	E	F	G	H	I
1	Budget		Year			Year			
2			Year=2010			Year=2011			
3	Category	Name	Qnty	Cost	Total	Qnty	Cost	Total	Total
4		travel	2	200	400	2	450	900	1300
5		hotel	5	100	500	8	80	640	1140
6		local travel	4	20	80	2	35	70	150
7	Total				980			1610	2590

Figure 1: Budget spreadsheet instance.

perform the evolution of spreadsheets. Section 5 discusses related work and Section 6 concludes the paper.

2 Motivating Example: Spreadsheet Refactoring

Suppose a researcher's yearly budget for travel and accommodation expenses is kept in the spreadsheet shown in Figure 1 taken from [11].

Note that throughout the years, cost and quantity are registered for three types of expenses (travel, hotel, local transportation). Formulas are used to calculate the total expense for each type in each year as well as the total expense in each year. Finally a grand total is calculated over all years, both per type of expense and overall.

At the end of 2010, this example spreadsheet needs to be modified to accommodate data 2011. A novice spreadsheet user would typically take four steps to perform the necessary modification:

- insert three new columns
- copy all the labels
- copy all the formulas (at least two)
- update all the necessary formulas in the last column

A more advanced user would shortcut these steps by copy-inserting the 3-column block of 2010 and changing the label "2010" to "2011" in the copied block. If the insertion is done behind the last year, the range of the multi-year totals columns must be extended to include the new year. If the insertion is done in between the last and one-but-last year, the spreadsheet environment automatically extends the formulas for the multi-year totals. Apart from these novice and advanced modification strategies, a mixed strategy may be employed. In any case, a conceptually unitary modification (*add year*) needs to be executed by a error-prone combination of steps.

Erwig *et al.* have introduced *ClassSheets* as models of spreadsheets that allow spreadsheet modifications to be performed at the right conceptual level. For example, the ClassSheet in Figure 2 provides a model of our budget spreadsheet.

In this model, the repetition of a block of columns for each year is captured by gray column labeled with the ellipsis. The horizontal repetition is marked in a analogous way. This makes it possible (i) to check whether the spreadsheet after modification still instantiates the same model, or (ii) to offer the user an unitary operation that extends the

	A	B	D	E	F	...	G
1	**Budget**		**Year**				
2			year=2010				
3	**Category**	Name	Qnty	Cost	Total		Total
4		name="abc"	qnty=0	cost=0	total=qnty*cost		total=SUM(total)
⋮							
5	Total				total=SUM(total)		total=SUM(**Year**.total)

Figure 2: Budget spreadsheet model.

E	F	G	H
Cost	Tax tarif	After tax	Total
cost=0	tax tarif=0	after tax=cost+ cost*tax tarif	total=qnty*cost
			total=SUM(total)

(a) New budget model.

D	E	F	G
Cost	Tax tarif	After tax	Total
200	0,12	224	400
100	0,2	120	500
20	0,2	24	80
			980

(b) New budget instance.

Figure 3: New spreadsheet and the model that it instantiates.

repetition with an extra block. Apart from (horizontal) block repetitions that support the extension with additional years, this model features (vertical) row repetitions that support the extension with new expense types.

Unfortunately, situations may occur in which the model itself needs to be modified. For example, if the researcher needs to report expenses before and after tax, additional columns need to be inserted in the block of each year. Figure 3 shows the new spreadsheet as well as the new model that it instantiates.

Note that a modification of the year block in the model (inserting various columns) captures modifications to all repetitions of the block throughout the instance.

In this paper, we will demonstrate that modifications to spreadsheet models can be supported by an appropriate combinator language, and that these model modifications can be propagated automatically to the spreadsheets that instantiate the models. In case of the budget example, the model modification is captured by the following expression:

$$
\begin{array}{l}
\mathit{addTax} = \mathit{once}\ (\\
\quad \mathit{inside}\ \texttt{"Year"}\ (\mathit{before}\ \texttt{"Total"}\ (\\
\quad\quad \mathit{insertCol}\ \texttt{"Tax Tariff"} \triangleright \mathit{insertCol}\ \texttt{"After tax"} \\
\quad))
\end{array}
$$

The actual column insertions are done by the innermost sequence of two $\mathit{insertCol}$ steps. The before and inside combinators specify the location constraints of applying these steps. The once combinator traverses the spreadsheet model to search for a single location where these constraints are satisfied and the insertions can be performed.

Application of the addTax transformation to the initial model (Figure 2) will yield:

- The modified model (Figure 3a)
- A spreadsheet migration function that can be applied to instances of the initial model (e.g. Figure 1) to produce instances of the modified model (e.g. Figure 3b)

- An inverse spreadsheet migration function to backport instances of the modified model to instances of the initial model

In the remainder of this paper, we will explain the machinery required for this type of coupled transformation of spreadsheet instances and models. As models, we will use a variation on ClassSheets where references are modeled by projection functions. Model transformations propagate references by composing instance-level transformations with these projection functions.

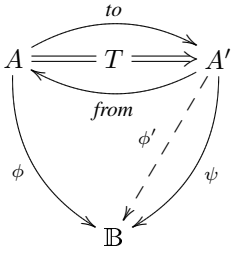
3 Two-level Coupled Transformations (2LT)

In this section we will revisit the two-level coupled transformations framework (2LT) and define the model used to represent spreadsheets.

3.1 Coupled transformations

The two-level data transformation (2LT) [3, 4, 5, 6], consists of type-level transformations coupled with data-value and/or function-level transformations. The type-level transformations deal with the evolution of the model, the data-value transformations deal the instances of the model (e.g. values), and the function-level transformations deal with the evolution of constraints and references.

Figure 4 depicts the general scenario of a transformation in 2LT with the invariants. The transformation T converts type A into type A'. Each transformation is coupled with witness functions to and $from$, which are responsible to convert values of type A into type A' and back. Also, the transformation might introduce for the new type A' a constraint ψ. Finally, type A might already have an constraint ψ, which should apply to the new type A'. This is done by introducing a new ψ' invariant by composing the $from$ function with the previous invariant. This is equivalent to converting the data instances to the original format and then apply the original constraint.

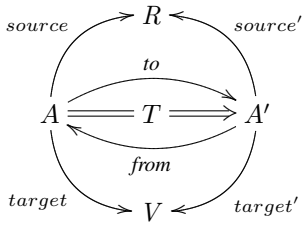


A, A'	datatype and transformed datatype
T	transformation of type A into type A'
to	witness function of type $A \rightarrow A'$ (injective)
$from$	witness function of type $A' \rightarrow A$ (surjective)
ϕ, ϕ'	constraint and transformed constraint
ψ	newly introduced constraint

$\phi' = \phi \circ from \wedge \psi$

Figure 4: Constraint-aware transformation of datatype A with constraint ϕ into datatype A' with constraint ϕ'. The constraint on the target type is the logical conjunction of (i) the constraint on the source type post-composed with the witness function $from$, and (ii) any new constraint ψ introduced by the type-change. When ϕ' is normalized it works on A' directly rather than via A.

Figure 5 depicts the scenario of a transformation with references. This scenario occurs simultaneously as a transformation with constraints, but is explained separately for simplicity. As in the previous case, transformation T converts type A into type A', which is coupled with witness functions to and $from$. A reference R to value V is defined using a pair of projections, $source$ and $target$. These projections are statically-typed functions, traversing the datatype A to identify the element in the datatype defining the reference, and the element to which the reference is pointing to. As analogy, this is comparable to the foreign key constraints defined in relational databases, defining the columns of a table that are foreign keys, and the tables and columns were the foreign key is defined. The advantage of this approach, is that not only references are statically typed, but also always guaranteed to exist.



A, A'	datatype and transformed datatype
R, V	reference and referenced value
T	transformation of type A into type A'
to, $from$	witness functions
$source$	Projection over type A identifying the reference
$target$	Projection over type A identifying the referenced value

$source' = source \circ from$
$target' = target \circ from$

Figure 5: Type-safe reference-aware transformation of datatype A into datatype A'. The projections defining the reference and the referenced type, in the transformed type A' are obtain by post-composing the projections with the witness function $from$. When $source'$ and $target'$ are normalized they work on A' directly rather than via A.

In summary, the 2LT provides the basic combinators to define and compose transformations for data-types, witness functions, constraints, and references. Since 2LT is statically typed, transformations are guarantee to be type-safe ensuring consistency of data-types, data instances, constraints and references.

3.2 Spreadsheet Model

In 2LT a spreadsheet is modeled using a Generalized Algebraic Data Type (GADT), based on the specification of the ClassSheet models [11]. The GADT [14] is defined as following:

```
data Type a where
...
  ·        :: Type a → PF (Pred a) → Type a                    -- invariants
...
  Value    :: Value → Type ·                                   -- plain value
  Ref      :: Type b → PF (a → RefCell) → PF (a → b)
              → Type a → Type a                                -- references
  RefCell  :: Type RefCell                                     -- reference cell
```

$$
\begin{array}{lll}
\cdot & :: \cdot \rightarrow \mathit{Type}\ \cdot & \text{-- formulas} \\
\mathit{LabelB} & :: \mathit{String} \rightarrow \mathit{Type}\ \mathit{LabelB} & \text{-- block label} \\
\cdot = \cdot & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a, b) & \text{-- attributes} \\
\cdot \shortmid \cdot & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a, b) & \text{-- block horizontal composition} \\
\cdot \hat{} \cdot & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a, b) & \text{-- block vertical composition} \\
\mathit{EmptyB} & :: \mathit{Type}\ \mathit{EmptyB} & \text{-- empty block} \\
\underline{\cdot} & :: \mathit{String} \rightarrow \mathit{Type}\ \mathit{HorH} & \text{-- horizontal class label} \\
\mid \cdot & :: \mathit{String} \rightarrow \mathit{Type}\ \mathit{VerV} & \text{-- vertical class label} \\
\mid \underline{\cdot} & :: \mathit{String} \rightarrow \mathit{Type}\ \mathit{Square} & \text{-- square class label} \\
\mathit{LabRel} & :: \mathit{String} \rightarrow \mathit{Type}\ \mathit{LabS} & \text{-- relation class} \\
\cdot : \cdot & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a, b) & \text{-- labeled class} \\
\cdot : \cdot^{\downarrow} & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a, [b]) & \text{-- labeled expandable class} \\
\cdot \hat{} \cdot & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a, b) & \text{-- class vertical composition} \\
\mathit{SheetC} & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ (\mathit{SheetC}\ a) & \text{-- sheet class} \\
\cdot^{\rightarrow} & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ [a] & \text{-- sheet expandable class} \\
\cdot \shortmid \cdot & :: \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a, b) & \text{-- sheet horizontal composition} \\
\mathit{EmptyS} & :: \mathit{Type}\ \mathit{EmptyS} & \text{-- empty sheet}
\end{array}
$$

The above GADT allows to formally specify a spreadsheet business model, references and constraints. In order to explain the GADT we will use as an example a reduced version of the budget spreadsheet model presented in Figure 1. For this reduced model only three columns were defined: quantity, cost per unit and total cost (computed by the product of quantity and cost per unit). In the first row, the labels are defined and the subsequent raws specify the values for the budget items. Using our GADT, this model is specified as following:

$$
\begin{array}{l}
\mathit{buy} = \mid \texttt{"Price List"} : \texttt{"Quantity"} \shortmid \texttt{"Price"} \shortmid \texttt{"Total"} \hat{} \\
\quad \texttt{"quantity"} = 0 \shortmid \texttt{"price"} = 0 \shortmid \texttt{"total"} = \mathit{FFormula}\ \texttt{"}\star\texttt{"}\ [\mathit{FRef}, \mathit{FRef}]^{\downarrow}
\end{array}
$$

In the first line we define a sheet containing a vertical class label called `"Price List"`. The vertical class, defined by the $\cdot : \cdot^{\downarrow}$ operator, is the initial piece of our model, which states that some blocks will be vertically repeated. The second line of the example specifies the first row of the spreadsheet table containing the headers (one per column as defined by the $\cdot \shortmid \cdot$ operator). The third example specifies the subsequent rows (through the $\cdot \hat{} \cdot$ operator) which defines the repeating three elements, one per column. These elements are `"Quantity"` defined as VInt (integer) with default value 0, `"Price"` defined as $\mathit{VDouble}$ (double) with defined value 0, and finally, `"Total"` which is a formula defined by the product between two references.

The formula specification, as previously shown is specified directly in the GADT. However, the references are defined separately by defining projections over the data type. This is required to allow any reference to access any type of the GADT. An example of reference from the formula to quantity, for the above example, is show as following:

$$
\begin{array}{l}
\mathit{withRefs} = \mathit{Ref}\ (0)\ \mathit{source}\ \mathit{target}\ \mathit{buy} \\
\quad \textbf{where}
\end{array}
$$

$$source = fhead \circ head \circ (\pi_2 \circ \pi_2 \circ \pi_2)^\star \circ \pi_2$$
$$target = head \circ (\pi_1 \circ \pi_2)^\star \circ \pi_2$$

Basically, a reference specifies two projections, a source and a target. The first projection, source, specifies where in the GADT the reference is defined. This refers to the first $FRef$ in our item cost specification (GADT). The second projection, target, defines the type it is pointing to, i.e. it defines a reference to the `"Quantity"` $= 0$ type. Since the use of GADTs requires the definition of models combining elements in a pairwise fashion, it is necessary to descend into the structure using π_1, π_2 and $\cdot^\star$.

Note that our reference type has enough information about the cells and so we do not need value-level values. In the cases we reference a list of values, for example, constructed by the class expandable operator, we need be specific about the element in the list we are referencing. For these cases we use the type-level constructors $head$ and $tail$ to get the intended value in the list.

4 Spreadsheets Evolution

In this section we define rules to perform spreadsheet evolution. These rules can be divided in two main categories: *Semantic* rules, intended to change the model itself (e.g. add a new column), and *Layout* rules, designed to change the visual arrangement of the spreadsheet (e.g. swap two columns). Tables 1 and 3 summarize the semantic and layout rules, respectively.

Semantic Rules	
$Block \leqslant Block \mid Block$	Add a block horizontally
$Block \leqslant Block \hat{} Block$	Add a block vertically
$Sheet \leqslant Sheet \mid Sheet$	Add a sheet
$Class \leqslant Class \hat{} Class$	Add a class
$Sheet \leqslant Sheet \mid Class$	Add a column
$Block \leqslant Block \mid Block$	Add a column
$Label : Class \leqslant Label : Class^{\downarrow}$	Make a block expandable
$Class \leqslant Class^{\rightarrow}$	Make a class expandable
$Sheet \leqslant (Sheet \mid Sheet)_{ref}$	Split
$Sheet \leqslant (Sheet \mid Sheet)_{ref}$	Split functional dependency

Table 1: Summary of semantic rules defined to transform spreadsheets.

To make the rules more flexible, they are defined as local as possible. Combinators are then used to apply these rules in specific places. In Table 2 the defined combinators are summarized.

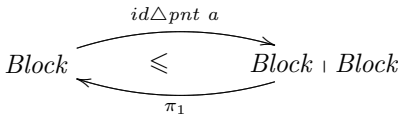
Combinators	
before/*after*/*below*/*above*	Apply a rule before/after/below/above something
once	Apply a rule once somewhere
inside	Apply a rule inside some given part
at	Apply a rule at a given part

Table 2: Summary of combinators defined to transform spreadsheets.

4.1 Semantic Rules

4.1.1 Insert a Block

One of the most fundamental rules is the insertion of a new block into a spreadsheet, formally defined as following.



This definition means that a horizontal composition of two blocks refines a block and it is witnessed by two functions. The *to* function, $id\triangle pnt\ a$, is a split: It injects the existing block in the first part of the result pair without modifications (id) and injects the given block instance a in the second part. The *from* function is π_1 since it is the one that allows the recovery of the existent block. The HASKELL version of the rule is presented next.

```
insertBlock :: Type a → a → Rule
insertBlock ta a tx | isBlock ta ∧ isBlock tx = do
  let rep = Rep { to = (id△pnt a), from = π1 }
  View s t ← pullUpAllRefs (tx ∣ ta)
  return (View (comprep (tx ∣ ta) rep s) t)
insertBlock _ _ _ = mzero
```

The first step of this rule is to create the representation with the witness functions *rep*, as explained in Section 3, which implementation follows the formal definition.

Its second step is to *pull up all the references.* To avoid having references in any level of the spreadsheet model, all our rules pull all the references to the top-most level of the model. This is achieved by applying the rule $pullUpAllRefs = many\ (once\ pullUpRef)$. A case of the rule $pullUpRef$ is defined below:

```
pullUpRef :: Rule
pullUpRef (Ref tb fRef tRef ta ∣ b2) = do
  return (View idrep (Ref tb (fRef ∘ π1) (tRef ∘ π1) (ta ∣ b2)))
 ...
```

If a branch (in this case the left one) of a given type has a reference, it is pulled to the top level. This rule has two cases for each spreadsheet binary constructor (left and

right). The combination of the combinators $once$ and $many$ guarantees that the rule is applied exhaustively in the given type.

The result type of $insertBlock$ is the one returned by $pullUpAllRefs$, which is the horizontal composition of the two blocks with all the references (if any) at the top-most level. The result representation function is the composition of the representation returned by $pullUpAllRefs$ and the representation rep.

Because the used representation is generic, we need to guarantee in the rules that the types they receive are the expected ones. We could have several cases in each function each one doing pattern matching in one of the correct constructors, but this would make our functions big. Instead, we decided to create functions that test the input by pattern matching in the correct constructors. This makes the code simpler and more readable. The function $isBlock$ tests if the received types are blocks. If non-blocks are passed as arguments, the rule fails because the test function also fails.

Rules to insert classes and sheets were also defined. However, since these rules are similar to the rule for inserting blocks, we have omitted them for brevity.

4.1.2 Insert a Column

To insert a column in a spreadsheet, that is, a cell with a label lbl and the cell bellow with a default value df and expandable vertically, we first need to create a new class representing it:

$$lbl' = map\ toLower\ lbl$$
$$clas =\mid lbl : lbl\hat{}(lbl' = df)^{\downarrow}$$

Using the label and the default value, we create a value-level value to be inserted:

$$d = (lbl, [(lbl, (lbl', df))])$$

Notice that, since we want to create an expandable class, the second part of the pair must be a list. The final step is to apply the $insertSheet$ rule:

$$\begin{array}{l} insertCol :: String \rightarrow VFormula \rightarrow Rule \\ insertCol\ l\ f@(FFormula\ name\ fs)\ tx \mid isSheet\ tx = \mathbf{do} \\ \quad \mathbf{let}\ lbl' = map\ toLower\ lbl \\ \qquad clas =\mid lbl : lbl\hat{}(lbl' = df)^{\downarrow} \\ \qquad d = (lbl, [(lbl, (lbl', df))]) \\ \quad ((insertSheet\ clas\ d) \rhd pullUpAllRefs)\ tx \end{array}$$

The case shown in the above definition is for a formula as default value and it is similar to the value case. The case with a reference is more interesting and is shown next:

$$\begin{array}{l} insertCol\ l\ FRef\ tx \mid isSheet\ tx = \mathbf{do} \\ \quad \mathbf{let}\ lbl' = map\ toLower\ lbl \\ \qquad clas =\mid lbl : Ref\ \bot\ \bot\ \bot\ (lbl\hat{}(lbl' = RefCell))^{\downarrow} \\ \qquad d = (lbl, [(lbl, \bot :: RefCell)]) \\ \quad ((insertSheet\ clas\ d) \rhd pullUpAllRefs)\ tx \end{array}$$

Recall that our references are always local, that is, they can only exist with the type they are associated with. So, it is not possible to insert a column that references a part of

the existing spreadsheet. To overcome this, we first create the reference with undefined functions and auxiliary type ($\perp$) and then we set these values to the intended ones.

```
setFormula :: Type b → PF (a → RefCell) → PF (a → b) → Rule
setFormula tb fromRef toRef (Ref _ _ _ t) =
  return (View idrep (Ref tb fromRef toRef t))
```

This rule receives the auxiliary type, the two functions and changes the reference accordingly.

A complete rule to insert a column with a reference could be defined as follows:

```
insertFormula = (once (insertCol "After Tax" FRef)) ▷
                  (setFormula auxType fromRef toRef)
```

Following the original idea described in Section 2, we want to introduce a new column with the tax tariff. In this case we want to insert a column in an existing block and thus our previous rule will not work. For these cases we wrote a new rule:

```
insertColIn :: String → VFormula → Rule
insertColIn l (FValue v) tx | isBlock tx = do
  let lbl' = map toLower lbl
      block = lbl^(lbl' = v)
      d = (lbl, v)
  ((insertBlock block d) ▷ pullUpAllRefs) tx
```

The rule is quite similar to the previous one but it creates a block (not a class) and inserts it also after a block. The reasoning is analogous to the one in *insertCol*.

To add the two columns `"Tax tariff"` and `"After tax"` we can use the rule *insertColIn*, but applying it directly to our running example will fail since it expects a block and we have a spreadsheet. We can use the combinator *once* to achieve the desired result (we assume that the column `"Tax tariff"` was already inserted):

```
ghci⟩let formula = FFormula "*" [FRef, FRef]
ghci⟩once (after "Tax tarif"
        (once (insertColIn "After Tax" formula))) budget
...
("Cost" | "Tax tariff" | "After tax"^("after tax" = formula) | "Total")^
("cost" = 0 | "tax tarif" = 0 | "total" = totalFormula)
...
```

The *once* combinator tries to apply a given rule somewhere in a type. It stops after it succeeds once. Although this combinator already existed in the 2LT framework, we had to extended it to work for spreadsheet models.

To allow to be specific about the right location of the application of the rules we developed a set of combinators. In the above example we use *after* which receives a string and a rule, and applies the rule immediately after the block with that string as label.

```
after :: String → Rule → Rule
after label r (l | ·@(label') a) | label ≡ label' = do
```

```
    View s l' ← r l
    let rep = Rep {to = to s × id, from = from s × id}
    return (View rep (l' ↓ a))
after _ _ _ = mzero
```

When it finds the intended place, it applies a rule to it. This works because our rules always do their task on the right of a type. More similar combinators were also developed: *before*, *bellow*, *above*, *inside* and *at*. Their implementation is not shown since they are similar to the *after* combinator.

Notice that the result from the above application of our rules is not quite right. The block inserted is a vertical composition and is inserted in a horizontal composition. The correct would be to have its top (bottom) part on the top (bottom) part of the result, as defined below:

```
("Cost" ↓ "Tax tariff" ↓ "After tax" ↓ "Total")^
("cost" = 0 ↓ "tax tarif" = 0 ↓ "after tax" = formula ↓ ·
  ("total" = totalFormula))
```

To correct these cases, we designed a layout rule *normalize*. This rule is explained in detail in Section 4.2.

4.1.3 Make it Expandable

It is possible to make a block in a class expandable. For this, we created a rule, *expandBlock*:

```
expandBlock :: String → Rule
expandBlock str (label : clas) | compLabel label str = do
  let rep = Rep {to = id × tolist, from = id × head}
  return (View rep (label : clas↓))
expandBlock _ _ = mzero
```

It receives the label of the class to make expandable and updates the class to allow repetition. The result type constructor if $\cdot : \cdot^{\downarrow}$ and the witness functions wrap the existing block into a list or take the head of it depending on the direction the rule is applied.

A similar rule was developed to the make a class expandable. This corresponds to promote a class c to $c^{\rightarrow}$. We do not show it here because its implementation is quite similar to this one.

4.1.4 Split

It is quite common to move a column in a spreadsheet from on place to another. The rule that we will present now copies a column to another place and substitute the current values by references to the new column. The rule to simply move a part of the spreadsheet is presented in Section 4.2. The first step is to get the column that we want to move out:

```
getColumn :: String → Rule
getColumn h t (l'^b1) | h ≡ l' = do
```

```
    return (View idrep t)
...
```

If the corresponding label is found, the vertical composition is returned. The representation *idrep* is a representation that has the *id* function in both directions. Notice that, as in many other rules, this rule is intended to be applied using the combinator *once*. As we said before, we aimed to write local rules that can be used at any level using the developed combinators.

In a second step, the rule create a new a class containing the retrieved block:

```
...
do View s c' ← getBlock str c
   let nsh =| str : c'↓
...
```

The last step is to transform the column that was copied out into references to the new column. The rule *makeReferences* :: *String* → *Rule* receives the label of the column that was copied out (the same as the new column) and creates the references. We do not shown the rest of the implementation because it is quite complex and will not help in the understanding of the paper.

Let us consider the following part of our example:

```
budget = ...
("Cost" ^| "Tax tariff" ^| "After tax" ^| "Total")^
("cost" = 0 ^| "tax tarif" = 0 ^| "after tax" = formula ^| ·
  ("total" = totalFormula))
...
```

If we apply the *split* rule to it we get the following:

```
ghci⟩once (split "Tax tariff") budget
...
("Cost" ^| "Tax tariff" ^| "After tax" ^| "Total")^
("cost" = 0 ^| "tax tarif" = 0 ^| RefCell ^| "total" = totalFormula)
^|
(| "Tax tariff" : ("Tax tariff"^
                   "tax tarif" = 0)↓)
```

4.1.5 Split Functional Dependencies

It is quite common to have columns with repeated data and inducing functional dependencies [7, 9, 10]. This duplication of data can be the source of errors and inconsistencies in the spreadsheet. A possible solution is to refact the spreadsheet creating a new table with the repeated information cleaned and substitute the existing columns by references to the new table.

A functional dependency can be represented by $A \rightharpoonup B$ where the set of columns A, the antecedent, functionally determines the set of columns B. This means that given a value in A we can determine its corresponding in B [20].

We defined and present a rule that automates this process, $splitFD :: [String] \rightarrow [String] \rightarrow Rule$.

Given two lists of column names, the antecedent and the consequent of the functional dependency, this rule creates a new class with the given columns and composes the existing sheet with the new class. At the value-level, the repeated data in the new table is removed. The old columns are all removed, except the ones that compose the antecedent of the functional dependency which are updated to references to the new table. This is possible because we know that there is a functional dependency imposing a relation between the antecedent and the consequent.

To discover functional dependencies in spreadsheets, the techniques presented in [7] can the used.

Although this is a complex operation, this will facilitate the maintenance of the spreadsheet. In fact we still can improve this result. In a database setting the relation that we create when we update the antecedent columns to references to the new table is called a *foreign key*, that is, a column (or a set of) that references another column (or set of columns). In the datbase realm this ensures that the foreign key column only contains existing values from another column.

Since 2LT has the expressibility of the invariants, we can add one invariant to this spreadsheet imposing that the column just updated to references just has references to the new column where the antecedent is, that is, a foreign key.

Moreover, given the flexibility of the invariants, we can add any kind of invariants to the spreadsheets. Referring again the database realm, a *primary key* is a set of columns that determine the other columns in a table, that is, no two equal values in the primary column can have different values in the other columns. A primary key induces a functional dependency. This can be easily added to a spreadsheet. Indeed the $splitFD$ rule ensures that the new table created has as primary key in the antecedent columns.

When exported to a spreadsheet application, for example, Excel, this invariants can easily be implemented as VBA scripts.

Allowing the integration of invariants, and in particular the ones related to the databases, the migration process to a database application becomes straightforward. For more on this subject, we refer the reader to the migration between spreadsheets and relation databases introduced in [10].

4.2 Layout Rules

In this section we describe rules focused on the layout of spreadsheets. In Table 3 we summarize the layout rules designed.

Notice that, since there is no changes in the data, these rules are isomorphisms ($\cong$).

4.2.1 Change Orientation

The rule $toVertical$ changes the orientation of a block from horizontal to vertical.

$$
\begin{aligned}
&toVertical :: Rule \\
&toVertical\ (a \mid b) = return\ (View\ idrep\ (a \hat{}\ b)) \\
&toVertical\ \_ = mzero
\end{aligned}
$$

Layout Rules	
$Sheet \cong Sheet$	Change orientation (to vertical/horizontal)
$Sheet \cong Sheet$	Normalize blocks
$Sheet \cong Sheet$	Shift part of the spreadsheet
$Sheet \cong Sheet$	Move part of the spreadsheet

Table 3: Summary of layout rules defined to transform spreadsheets.

Note that, since our value-level representation of these compositions are pairs, the to and the $from$ functions are simply the identity function. The needed information is kept in the type-level with the different constructors.

A rule to do the inverse was also designed but since it is quite similar to this one, we do not show it here.

4.2.2 Normalize Blocks

When applying some transformations, the resulting types my not be in the exactly correct shape. A common example is to have as result the following type:

$A \mid B \hat{} C \mid D \hat{}$
$E \mid F$

Most of the times, the correct result is as described bellow:

$A \mid B \mid D \hat{}$
$E \mid C \mid F$

The rule $normalize$ tries to match these cases and correct them. The types are the ones presented above and the witness functions are combinations of π_2 and π_1.

$$
\begin{aligned}
&normalize1 :: Rule \\
&normalize1\ (a \mid b \hat{} c \mid d \hat{} e \mid f) = \\
&\quad return\ (View\ (Rep\ \{to = tof, from = fromf\})\ (a \mid b \mid d \hat{} e \mid c \mid f)) \\
&\quad \textbf{where} \\
&\quad\quad tof = id \times \pi_1 \times id \circ \pi_1 \triangle \pi_1 \circ \pi_2 \triangle \pi_2 \circ \pi_1 \circ \pi_2 \times \pi_2 \\
&\quad\quad fromf = \pi_1 \circ \pi_1 \triangle \pi_1 \circ \pi_2 \times \pi_1 \circ \pi_2 \triangle \pi_2 \circ \pi_2 \circ \pi_1 \triangle id \times \pi_2 \circ \pi_2 \\
&normalize1\ \_ = mzero
\end{aligned}
$$

Although the migration functions seem complex, they just rearrange the order of the pair so they have the correct order.

4.2.3 Shift

It is quite common to move parts of the spreadsheet across it. We designed a rule to shift pieces of the spreadsheet in the four possible directions. We show here part of the $shitRight$ rule, which, as suggested by its name, shifts a piece of the spreadsheet to the right. In this case, a block is moved and an empty block is left in its place.

```
shitRight :: Type a → Rule
shitRight ta b1 | isBlock b1 = do
  Eq ← teq ta b1
  return (View (Rep {to = pnt (⊥ :: EmptyBlock)△id, from = π2}) (EmptyBlock ▵ b1))
```

Notice that this rule receives a type, a block, but we can easily write a wrapper function to receive a label in the same style of $insertCol$ for example.

Another interesting case of this rules occurs when the user tries to move a block (or a sheet) that has a reference.

```
shitRight ta (Ref tb frRef toRef b1) | isBlock2 b1 = do
  Eq ← teq ta b1
  let rep = Rep {to = pnt (⊥ :: EmptyBlock)△id, from = π2}
  return (View rep (Ref tb (frRef ∘ π2) (toRef ∘ π2) (EmptyBlock ▵ b1)))
```

As we can see in the above code, the existing reference formulas must be composed with the selector π_2 to allow to retrieve the existing block $b1$. Only after this it is possible to apply the defined selection reference functions.

4.2.4 Move Blocks

A more complex rule is to move a part of the spreadsheet to another place. We present next a rule to move a block.

```
moveBlock :: String → Rule
moveBlock str c =
  do View s c' ← getBlock str c
     let nsh =| str : c'
     View r sh ← once (removeRedundant str) (c ▵ nsh)
     return (View (comprep s r) sh)
moveBlock _ _ = mzero
```

After getting the intended block and creating a new class with it, we need to remove the old block using $removeRedundant$.

```
removeRedundant :: String → Rule
removeRedundant s (s') | s ≡ s' = return (View rep EmptyBlock)
  where
    rep = Rep {to = pnt (⊥ :: EmptyBlock), from = pnt s'}
removeRedundant _ _ = mzero
```

This rule will remove the block with the given label leaving an empty block in its place.

5 Related Work

Ko et al. [15] summarizes and classifies the research challenges of the End-User software engineering area. These include requirements gathering, design, specification, reuse, testing and debugging. Ko et al. recognizes the importance of testing and model

validation to prevent the introduction of errors during the design phase. However, besides the importance of Lehman's laws of software evolution [18], very little is stated with respect to spreadsheet evolution. Spreadsheets evolution poses challenges not only in the evolution of the underlying model, but also in migration of the spreadsheet values and used formulæ. Nevertheless, many of the underlaying transformations used for spreadsheet transformations are shared with works for spreadsheet generation and other program transformation techniques.

Engels et al. [19] propose a first attempt to solve the problem of spreadsheet evolution. ClassSheets are used to specify the spreadsheet model and transformation rules are defined to enable model evolution. These model transformations are propagated to the model instances (spreadsheets), through a second set of rules which updates the spreadsheet values. Engels et al. present a set of rules and a prototype of tool to support these changes. In this paper we present a more advanced way to evolve spreadsheets models and instances differing from the one presented by Engels et al. in several ways. First, we use strategic programming with two-level coupled transformation (2LT). This enables type-safe transformations, offering guarantee that in any step semantics are preserved. Also, the use of 2LT not only gives for free the data migration but also it allows back portability, i.e. allows the migration of data from the new model back to the old model. Finally, we treat constraints in a formal way both in terms of specification and transformation, evolving constraints with the data model and data migration.

The authors [8] have proposed a methodology to transform spreadsheets into relational databases and back. This methodology is based on data refinements using bidirectional transformations. A generic abstract data type (GADT) augmented with constraints is used to define both the relational and spreadsheet model. Data mining rules are used to discover functional dependencies of data in order to infer an optimized model eliminating data redundancy. Although the same transformation principles are shared, the contributions of this work are much different. While in [8] the contribution migration from a spreadsheet with flat-structure to a relational model in a database. In this work we solve the problem of co-evolution of spreadsheet models and model instances (spreadsheets themselves), which poses new challenges with respect to the transformation rules and defined constraints.

The authors and other [3] proposed a solution for constraint-aware schema transformations also using 2LT. A generic abstract data type (GADT) was defined to represent schemas and schema constraints (such as foreign key relationships). Rules were defined to automatically evolve the schemas, schema constraints and the migration of data. The contribution of this work differs from the one in [3]. In this paper we do spreadsheet evolution, instead of simple algebraic evolution. Additionally, the set of transformation rules used for spreadsheet evolution are new.

Vermolen and Visser [24] propose a different approach for coupled evolution of data model and data. From a data model definition, they generate a domain specific language (DSL) which supports the basic transformations and allowing data model and data evolution. The interpreter for the DSL is automatically generated making this approach operational. This approach could also be used for spreadsheet evolution. However there are a few important differences. While the approach from Vermolen and Visser are tailored for forward evolution, our approach supports reverse engineering, i.e. supports automatic transformation and migration from the new model to the

old model. Also, our approach additionally takes into account the evolution of data constraints.

The importance of the co-evolution of different software artifacts has been reported before. Lämmel [16] first identified this category of transformations and showed and its applicability to a widespread and diversity of problems. Finally, Favre [13] established the importance of co-evolution not only between software languages and programs, but also for meta-models and models.

6 Conclusions

In this paper, we have presented an approach for disciplined model-driven evolution of spreadsheets. The approach takes as starting point the observation that spreadsheets can be seen as instances of a spreadsheet model where the model captures the business logic of the spreadsheet. We have extended the calculus for coupled transformations of the 2LT platform to this spreadsheet model. An important novel aspect of this extension is the treatment of references. In particular, we have made the following contributions:

- We have provided a model of spreadsheets in the form of a generalized algebraic datatype with embedded point-free function representations. This model is reminiscent of the ClassSheet models of Erwig *et al* [11].
- We have defined a coupled transformation system in which transformations at the level of spreadsheet models are coupled with corresponding transformations at the level of spreadsheet data/instances. The transformation system combines strategy combinators known from strategic programming [17] with spreadsheet-specific transformation rules.
- We have illustrated our approach with a number of specific spreadsheet refactorings to perform the evolution of spreadsheets.

In this paper we have presented our techniques in the context of spreadsheet models. As future work, we will study how to generalize it to other kinds of models where references play a role such as object-oriented modeling languages, e.g. UML class diagrams.

Acknowledgment

The first and the third authors are supported by the *Fundação para a Ciência e a Tecnologia* grants SFRH/BD/30231/2006 and SFRH/BD/30215/2006, respectively.

This work was supported by the SSaaPP project (SpreadSheets as a Programming Paradigm) under FCT contract PTDC/EIA-CCO/108613/2008.

References

[1] Abraham, R., Erwig, M.: Inferring templates from spreadsheets. In: ICSE '06: Proceedings of the 28th international conference on Software engineering. pp. 182–191. ACM, New York, NY, USA (2006)

[2] Abraham, R., Erwig, M., Kollmansberger, S., Seifert, E.: Visual specifications of correct spreadsheets. In: VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 189–196. IEEE Computer Society, Washington, DC, USA (2005)

[3] Alves, T., Silva, P., Visser, J.: Constraint-aware Schema Transformation. In: The Ninth International Workshop on Rule-Based Programming (2008)

[4] Cunha, A., Oliveira, J., Visser, J.: Type-safe two-level data transformation. In: Misra, J., et al. (eds.) Proc. Formal Methods, 14th Int. Symp. Formal Methods Europe. LNCS, vol. 4085, pp. 284–299. Springer (2006)

[5] Cunha, A., Visser, J.: Strongly typed rewriting for coupled software transformation. ENTCS 174(1), 17–34 (2007), 7th Workshop on Rule-Based Programming

[6] Cunha, A., Visser, J.: Transformation of structure-shy programs: applied to XPath queries and strategic functions. In: Ramalingam, G., Visser, E. (eds.) Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007. pp. 11–20. ACM (2007), http://doi.acm.org/10.1145/1244381.1244385

[7] Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: 2010 IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE (2010), to appear

[8] Cunha, J., Saraiva, J., Visser, J.: From spreadsheets to relational databases and back. In: PEPM '09: Proceedings of the 2009 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 179–188. ACM, New York, NY, USA (2008)

[9] Cunha, J., Saraiva, J., Visser, J.: Discovery-based edit assistance for spreadsheets. In: 2009 IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 233–237. IEEE (2009)

[10] Cunha, J., Saraiva, J., Visser, J.: From spreadsheets to relational databases and back. In: PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation. pp. 179–188. ACM, New York, NY, USA (2009)

[11] Engels, G., Erwig, M.: ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In: Redmiles, D., Ellman, T., Zisman, A. (eds.) 20th IEEE/ACM Int. Conf. on Automated Sof. Eng., Long Beach, USA. pp. 124–133. ACM (2005)

[12] EuSpRIG: European spreadsheet risks interest group. http://www.eusprig.org/

[13] Favre, J.M.: Meta-model and model co-evolution within 3d software space. In: In Procs. of the Int. Workshop on. Evolution of Large-scale Industrial Software Applications (September 2003)

[14] Hinze, R.: Fun with phantom types. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming, pp. 245–262. Palgrave (2003)

[15] Ko, A., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrence, J., Lieberman, H., Myers, B., Rosson, M., Rothermel, G., Shaw, M., Wiedenbeck, S.: The state of the art in end-user software engineering. Journal ACM Computing Surveys (2009)

[16] Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: First International Workshop on Software Evolution Transformations (Nov 2004)

[17] Lämmel, R., Visser, E., Visser, J.: The Essence of Strategic Programming (Oct8 2003)

[18] Lehman, M.M.: Laws of software evolution revisited. In: EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology. pp. 108–124. Springer-Verlag, London, UK (1996)

[19] Luckey, M., Erwig, M., Engels, G.: Systematic evolution of typed (model-based) spreadsheet applications, submitted for publication

[20] Maier, D.: The Theory of Relational Databases. Computer Science Press (1983)

[21] Panko, R.: Spreadsheet errors: What we know. what we think we can do. Proceedings of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (EuSpRIG) (July 2000)

[22] Powell, S.G., Baker, K.R.: The Art of Modeling with Spreadsheets. John Wiley & Sons, Inc., New York, NY, USA (2003)

[23] Rajalingham, K., Chadwick, D., Knight, B.: Classification of spreadsheet errors. European Spreadsheet Risks Interest Group (EuSpRIG) (2001)

[24] Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008). Lecture Notes in Computer Science, vol. 5301, pp. 630–644. Springer, Heidelberg (September 2008)