Type-Safe Evolution of Spreadsheets

Jácome Cunha[1,2,⋆], Joost Visser[3], Tiago Alves[1,3,⋆⋆], and João Saraiva[1]

[1] Universidade do Minho, Portugal
{jacome,jas}@di.uminho.pt
[2] ESTGF, Instituto Politécnico do Porto, Portugal
[3] Software Improvement Group, The Netherlands
{j.visser,t.alves}@sig.eu

Abstract. Spreadsheets are notoriously error-prone. To help avoid the introduction of errors when changing spreadsheets, models that capture the structure and interdependencies 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.

We have designed an appropriate representation of spreadsheet models, including the fundamental notions of formulæand references. For these models and their instances, we have designed coupled transformation rules that cover specific spreadsheet evolution steps, such as the 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 inducing compound transformations at the instance level. This approach guarantees safe evolution of spreadsheets even when models change.

1 Introduction

Spreadsheets are widely used by non-professional programmers, the so-called *end users*, to develop business applications. Spreadsheet systems offer end users a high level of flexibility, making it easier to get started working with them. 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 [19,21,22].

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 end-users productivity, 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 [8,11], and the inclusion of visual objects to provide editing assistance in spreadsheets [10]. All these approaches propose a form of

⋆ Supported by Fundação para a Ciência e a Tecnologia, grant no. SFRH/BD/30231/2006.
⋆⋆ Supported by Fundação para a Ciência e a Tecnologia, grant no. SFRH/BD/30215/2006. Work supported by the SSaaPP project, FCT contract no. PTDC/EIA-CCO/108613/2008.

D. Giannakopoulou and F. Orejas (Eds.): FASE 2011, LNCS 6603, pp. 186–201, 2011.

end user model-driven software development: a spreadsheet business model is defined, from which then a customized spreadsheet application is generated guaranteeing the consistency of the spreadsheet data with the underlying model. In a recent empirical study we have shown that the use of model-based spreadsheets do improve end-users productivity [7].

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 [8] 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 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 framework [4].

In this paper we present an appropriate representation of a spreadsheet model, based on the ClassSheet business model, 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. We have implemented this technique in the HAEXCEL framework (available from the first author's web page: `http://www.di.uminho.pt/~jacome`): a set of HASKELL-based libraries and tools to manipulate spreadsheets. 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 describe the framework to model and manipulate spreadsheets. Section 4 defines the rules to 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 and 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 spreadsheet needs to be modified to accommodate 2011 data. A novice spreadsheet user would typically take four steps to perform the necessary task:

	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

Fig. 1. Budget spreadsheet instance

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 system automatically extends the formulas for the multi-year totals. Apart from these two strategies, a mixed strategy may be employed. In any case, a conceptually unitary modification (*add year*) needs to be executed by an 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.

	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)

Fig. 2. Budget spreadsheet model

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, and (ii) to offer the user an unitary operation. Apart from (horizontal) block repetitions that support the extension with more 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:

Cost	Tax tariff	After tax	Total
cost=0	tax tariff=0	after tax=cost+ cost*tax tariff	total=qnty*cost
			total=SUM(total)

(a) New budget model

Cost	Tax tariff	After tax	Total
200	0,12	224	400
100	0,2	120	500
20	0,2	24	80
			980

(b) New budget instance

Fig. 3. New spreadsheet and the model that it instantiates

$$addTax = once\ (inside\ \texttt{"Year"}\ (before\ \texttt{"Total"}\ (\\ insertCol\ \texttt{"Tax Tariff"} \triangleright insertCol\ \texttt{"After tax"})))$$

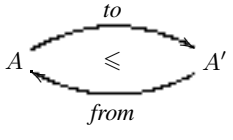
The actual column insertions are done by the innermost sequence of two $insertCol$ steps. The $before$ and $inside$ combinators specify the location constraints of applying these steps. The $once$ combinator traverses the spreadsheet model to search for a single location where these constraints are satisfied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

Application of the $addTax$ transformation to the initial model (Figure 2) will yield: firstly, the modified model (Figure 3a), secondly 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), and thirdly 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 A Framework for Evolution of Spreadsheets in HASKELL

Data refinement theory provides an algebraic framework for calculating with data types and corresponding values [16,17,18]. It consists of type-level coupled with value-level transformations. The type-level transformations deal with the evolution of the model and the value-level transformations deal with the instances of the model (*e.g.* values). Figure 4 depicts the general scenario of a transformation in this framework.



A, A' data type and transformed data type
to witness function of type $A \to A'$ (injective)
$from$ witness function of type $A' \to A$ (surjective)

Fig. 4. Coupled transformation of data type A into data type A'

Each transformation is coupled with witness functions to and $from$, which are responsible for converting values of type A into type A' and back.

The 2LT framework is an HASKELL implementation of this theory [3,4,5,6]. It provides the basic combinators to define and compose transformations for data types and witness functions. Since 2LT is statically typed, transformations are guaranteed to be type-safe ensuring consistency of data types and data instances.

3.1 ClassSheets and Spreadsheets in HASKELL

The 2LT was originally designed to work with algebraic data types. However, this representation is not expressive enough to represent ClassSheet specifications or their spreadsheet instances. To overcome this issue, we extended the 2LT representation so it could support ClassSheet models, by introducing the following *Generalized Algebraic Data Type*[1] (GADT) [12,20]:

```
data Type a where
  ...
  Value    :: Value → Type Value                                   -- plain value
                                                                   -- references
  Ref      :: Type b → PF (a → RefCell) → PF (a → b) → Type a → Type a
  RefCell  :: Type RefCell                                         -- reference cell
  Formula  :: Formula → Type Formula                               -- formulas
  LabelB   :: String → Type LabelB                                 -- block label
  · = ·    :: Type a → Type b → Type (a, b)                        -- attributes
  · | ·    :: Type a → Type b → Type (a, b)      -- block horizontal composition
  · ^ ·    :: Type a → Type b → Type (a, b)      -- block vertical composition
  EmptyB   :: Type EmptyB                                          -- empty block
  _        :: String → Type HorH                                   -- horizontal class label
  | ·      :: String → Type VerV                                   -- vertical class label
  | _      :: String → Type Square                                 -- square class label
  LabRel   :: String → Type LabS                                   -- relation class
  · : ·    :: Type a → Type b → Type (a, b)                        -- labeled class
  · : (·)↓ :: Type a → Type b → Type (a, [b])    -- labeled expandable class
  · ^ ·    :: Type a → Type b → Type (a, b)      -- class vertical composition
  SheetC   :: Type a → Type (SheetC a)                             -- sheet class
  ·→       :: Type a → Type [a]                  -- sheet expandable class
  · | ·    :: Type a → Type b → Type (a, b)      -- sheet horizontal composition
  EmptyS   :: Type EmptyS                                          -- empty sheet
```

The comments should clarify what the constructors represent. The values of type $Type\ a$ are representations of type a. For example, if t is of type $Type\ Value$, then t represents the type $Value$. The following types are needed to construct values of type $Type\ a$:

```
data EmptyBlock                                                    -- empty block
data EmptySheet                                                    -- empty sheet
type LabelB = String                                               -- label
data RefCell = RefCell1                                            -- referenced cell
type LabS = String                                                 -- square label
type HorH = String                                                 -- horizontal label
type VerV = String                                                 -- vertical label
data SheetC a = SheetCC a                                          -- sheet class
data SheetCE a = SheetCEC a                                        -- expandable sheet class
data Value = VInt Int | VString String | VBool Bool | VDouble Double   -- values
data Formula1 = FValue Value | FRef | FFormula String [Formula1]   -- formula
```

[1] "It allows to assign more precise types to data constructors by restricting the variables of the datatype in the constructors' result types."

Once more, the comments should clarify what each type represents.

To explain this representation we will use as an example a reduced version of the budget model presented in Figure 1. For this reduced model only three columns were defined: *quantity*, *cost per unit* and *total cost* (product of *quantity* by *cost per unit*).

$$
\begin{aligned}
&purchase = \\
&\quad | \; Price\_List : Quantity \shortmid Price \shortmid Total \hat{} \\
&\quad | \; PriceList : (quantity = 0 \shortmid price = 0 \shortmid total = FFormula \;\times\; [FRef, FRef])^{\downarrow}
\end{aligned}
$$

This ClassSheet specifies a class called $Price\_List$ composed by two parts vertically composed as indicated by the $\hat{}$ operator. The first part is defined in the first row and defines the labels for three columns: $Quantity$, $Price$ and $Total$. The second row defines the rest of the class containing the definition of the three columns. The first two columns have as default value 0 and the third is defined by a formula (explained latter on). Note that this part is vertical expandable, that is, it can be vertically repeated. In a spreadsheet instance this corresponds to the possibility of adding new rows. Figure 5 represents a spreadsheet instance of this model.

	A	B	C
1	Quantity	Price	Total
2	2	1500	=A2*B2
3	5	45	=A3*B3

Fig. 5. Spreadsheet instance of the $purchase$ ClassSheet

Note that in the definition of $Type\ a$ the constructors combining parts of the spreadsheet (*e.g.* sheets) return a pair. Thus, a spreadsheet instance is written as nested pairs of values. The spreadsheet illustrated in Figure 5 is encoded in HASKELL as follows:

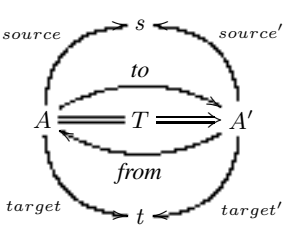
$$
\begin{aligned}
&((Quantity, (Price, Total \qquad\qquad\quad)), \\
&[(2 \quad\;\; , (1500 \;, FormulaFF \;\times\; [FRef, FRef])), \\
&\;(5 \quad\;\; , (45 \quad , FormulaFF \;\times\; [FRef, FRef]))])
\end{aligned}
$$

The HASKELL type checker statically ensures that the pairs are well formed and are constructed in the correct order.

3.2 Specifying Formulas

Having defined a GADT to represent ClassSheet models, we need now a mechanism to define spreadsheet formulas. The safer way to specify formulas is making them strongly typed. Figure 6 depicts the scenario of a transformation with references. A reference from a cell s to the a cell t is defined using a pair of projections, $source$ and $target$. These projections are statically-typed functions traversing the data type A to identify the cell defining the reference (s), and the cell to which the reference is pointing to (t). In this approach, not only the references are statically typed, but also always guaranteed to exist, that is, one can not create a reference from/to a cell that does not exist.

The projections defining the reference and the referenced type, in the transformed type A', are obtained by post-composing the projections with the witness function $from$.



$source$ Projection over type A identifying the reference
$target$ Projection over type A identifying the referenced cell

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

Fig. 6. Coupled transformation of data type A into data type A' with references

When $source'$ and $target'$ are normalized they work on A' directly rather than via A. 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 part of the GADT.

Using the spreadsheet illustrated in Figure 5, an instance of a reference from the formula *total* to *price* is defined as follows (remember that the second argument of Ref is the source (reference cell) and that the third is the target (referenced cell)):

$$purchaseWithReference = \\ Ref\ Int\ (fhead \circ head \circ (\pi_2 \circ \pi_2)^\star \circ \pi_2)\ (head \circ (\pi_1 \circ \pi_2)^\star \circ \pi_2)\ purchase$$

The $source$ function refers to the first $FRef$ in the HASKELL encoding shown after Figure 5. The $target$ projection defines the cell it is pointing to, that is, it defines a reference to the the value 1500 in column $Price$. 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 and π_2. The operator $\cdot^\star$ applies a function to all the element of a list and $fhead$ gets the first reference in a list of references.

Note that our reference type has enough information about the cells and so we do not need value-level functions, that is, we do not need to specify the projection functions themselves, just their types. In the cases we reference a list of values, for example, constructed by the class expandable operator, we need be specific about the element within the list we are referencing. For these cases, we use the type-level constructors $head$ (first element of a list) and $tail$ (all but first) to get the intended value in the list.

3.3 Rewriting Systems

At this point we are now able to represent ClassSheet models, including formulas. In this section we discuss the definition of the witness functions *from* and to. Once again we rely on the definition of a GADT:

$$\begin{array}{ll}
\textbf{data}\ PF\ a\ \textbf{where} & \\
\quad id & :: PF\ (a \to a) \qquad \text{-- identity function} \\
\quad \pi_1 & :: PF\ ((a,b) \to a) \qquad \text{-- left projection of a pair} \\
\quad \pi_2 & :: PF\ ((a,b) \to b) \qquad \text{-- right projection of a pair} \\
\quad pnt & :: a \to PF\ (One \to a) \qquad \text{-- constant} \\
\quad \cdot \triangle \cdot & :: PF\ (a \to b) \to PF\ (a \to c) \to PF\ (a \to (b,c)) \qquad \text{-- split of functions} \\
\quad \cdot \times \cdot & :: PF\ (a \to b) \to PF\ (c \to d) \to PF\ ((a,c) \to (b,d)) \qquad \text{-- product of functions}
\end{array}$$

$$
\begin{array}{lll}
\cdot \circ \cdot & :: Type\ b \to PF\ (b \to c) \to PF\ (a \to b) \to PF\ (a \to c) & \text{-- composing func.} \\
\cdot^{\star} & :: PF\ (a \to b) \to PF\ ([a] \to [b]) & \text{-- map of functions} \\
head & :: PF\ ([a] \to a) & \text{-- head of a list} \\
tail & :: PF\ ([a] \to [a]) & \text{-- tail of a list} \\
fhead & :: PF\ (Formula1 \to RefCell) & \text{-- head of the arguments of a formula} \\
ftail & :: PF\ (Formula1 \to Formula1) & \text{-- tail of the arguments of a formula}
\end{array}
$$

This GADT represents the types of the functions used in the transformations. For example, π_1 represents the type of the function that projects the first part of a pair. The comments should clarify which function each constructor represents. Given these representations of types and functions, we can turn to the encoding of refinements. Each refinement is encoded as a two-level rewriting rule:

$$\mathbf{type}\ Rule = \forall\ a\ .\ Type\ a \to Maybe\ (View\ (Type\ a))$$

$$\mathbf{data}\ View\ a\ \mathbf{where}\ View :: Rep\ a\ b \to Type\ b \to View\ (Type\ a)$$

$$\mathbf{data}\ Rep\ a\ b = Rep\ \{to = PF\ (a \to b), from = PF\ (b \to a)\}$$

Although the refinement is from a type a to a type b, this can not be directly encoded since the type b is only known when the transformation completes, so the type b is represented as a *view* of the type a. A *view* expresses that a type a can be represented as a type b, denoted as $Rep\ a\ b$, if there are functions $to :: a \to b$ and $from :: b \to a$ that allow data conversion between one and the other. The following code implements a rule to transform a list into a map (represented by $\cdot \rightharpoonup \cdot$):

$$
\begin{array}{l}
listmap :: Rule \\
listmap\ ([a]) = Just\ (View\ (Rep\ \{to = seq2index, from = tolist\})\ (Int \rightharpoonup a)) \\
listmap\ \_ = mzero
\end{array}
$$

The witness functions have the following signature (their code here is not important):

$$tolist :: (Int \rightharpoonup a) \to [a] \qquad seq2index :: [a] \to Int \rightharpoonup a$$

This rule receives the type of a list of a, $[a]$, and returns a view over the type map of integers to a, $Int \rightharpoonup a$. The witness functions are returned in the representation Rep. If other argument than a list is received, then the rule fails returning $mzero$. All the rules contemplate this case and so we will not show it in the definition of other rules.

Given this encoding of individual rewrite rules, a complete rewrite system can be constructed via the following constructors:

$$
\begin{array}{ll}
nop :: Rule & \text{-- identity} \\
\triangleright :: Rule \to Rule \to Rule & \text{-- sequential composition} \\
\oslash :: Rule \to Rule \to Rule & \text{-- left-biased choice} \\
many :: Rule \to Rule & \text{-- repetition} \\
once :: Rule \to Rule & \text{-- arbitrary depth rule application}
\end{array}
$$

Details on the implementation of these combinators can be found elsewhere [4].

4 Spreadsheets Evolution

In this section we define rules to perform spreadsheet evolution. These rules can be divided in three main categories: *Combinators*, used as helper rules, *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).

4.1 Combinators

The other types of rules are defined to work on a specific part of the data type. The combinators defined next are then used to apply those rules in the desired places.

Pull Up All the References: To avoid having references in different levels of the models, all the rules pull all the references to the topmost level of the model. To pull a reference is a particular place we use the following rule (we show just its first case):

$$
\begin{aligned}
&pullUpRef :: Rule \\
&pullUpRef\ ((Ref\ tb\ fRef\ tRef\ ta) \mid b2) = \mathbf{do} \\
&\quad return\ (View\ idrep\ (Ref\ tb\ (fRef \circ \pi_1)\ (tRef \circ \pi_1)\ (ta \mid b2)))
\end{aligned}
$$

The representation $idrep$ has the id function in both directions. If part of the model (in this case the left part of a horizontal composition) of a given type has a reference, it is pulled to the top level. This is achieved by composing the existing projections with the necessary functions, in this case π_1. This rule has two cases (left and right hand side) for each binary constructor (*e.g.* horizontal/vertical composition).

To pull up all the references in all levels of a model we use the rule $pullUpAllRefs = many\ (once\ pullUpRef)$. The $once$ operator applies the $pullUpRef$ rule somewhere in the type and the $many$ ensures that this is applied everywhere in the whole model.

Apply After and Friends: The combinator $after$ finds the correct place to apply the argument rule (second argument) by comparing the given string (first argument) with the existing labels in the model. When it finds the intended place, it applies the rule to it. This works because our rules always do their task on the right-hand side of a type.

$$
\begin{aligned}
&after :: String \to Rule \to Rule \\
&after\ label\ r\ (label' \mid a) \mid label \equiv label' = \mathbf{do} \\
&\quad View\ s\ l' \leftarrow r\ label' \\
&\quad return\ (View\ (Rep\ \{to = to\ s \times id, from = from\ s \times id\})\ (l' \mid a))
\end{aligned}
$$

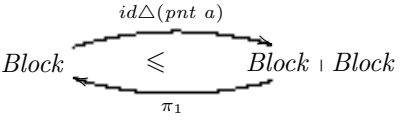
Note that this definition is only part of the complete version since it only contemplates the case for horizontal composition of blocks $(\cdot \mid \cdot)$.

Other combinators were also developed, namely, $before$, $bellow$, $above$, $inside$ and at. Their implementations are not shown since they are similar to the $after$ combinator.

4.2 Semantic Rules

In this section we present rules that change the semantics of the model, for example, adding columns.

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



This diagram means that a horizontal composition of two blocks refines a block when witnessed by two functions, to and $from$. The to function, $id\triangle(pnt\ a)$, is a split: it injects the existing block in the first part of the result without modifications (id) and injects the given block instance a into the second part of the result. 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.

$$
\begin{array}{l}
insertBlock :: Type\ a \rightarrow a \rightarrow Rule \\
insertBlock\ ta\ a\ tx \mid isBlock\ ta \wedge isBlock\ tx = \mathbf{do} \\
\quad \mathbf{let}\ rep = Rep\ \{to = (id\triangle(pnt\ a)), from = \pi_1\} \\
\quad View\ s\ t \leftarrow pullUpAllRefs\ (tx \shortmid ta) \\
\quad return\ (View\ (comprep\ rep\ s)\ t)
\end{array}
$$

The function $comprep$ composes two representations. This rule receives the type of the new block ta, its default instance a, and returns a $Rule$. The returned rule is itself a function that receives the block to modify tx and returns a view of the new type. The first step is to verify if the given types are block using the function $isBlock$. The second step is to create the representation rep with the witness functions given in the above diagram. Then the references are pulled up in result type $tx \shortmid ta$. This returns a new representation s and a new type t (in fact, the type is the same $t = tx \shortmid ta$). The result view has as representation the composition of the two previous representations, rep and s, and the corresponding type t.

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

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 vertically expandable, we first need to create a new class representing it: $clas =\mid lbl : lbl\hat{}(lbl = df^{\downarrow})$. The label is used to create the default value $(lbl, [\,])$. Note 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 $insertSheet$:

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

Note the use of the rule $pullUpAllRefs$ as explained before. 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:

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

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 ($\bot$) and then we set these values to the intended ones.

$$
\begin{array}{l}
setFormula :: Type\ b \rightarrow PF\ (a \rightarrow RefCell) \rightarrow PF\ (a \rightarrow b) \rightarrow Rule \\
setFormula\ tb\ fRef\ tRef\ (Ref\ \_\ \_\ \_\ t) = return\ (View\ idrep\ (Ref\ tb\ fRef\ tRef\ t))
\end{array}
$$

This rule receives the auxiliary type ($Type\ b$), the two functions representing the reference projections and adds them to the type. A complete rule to insert a column with a reference is 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 write a new rule:

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

This rule is 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. This combinator tries to apply a given rule somewhere in a type, stopping after it succeeds once. Although this combinator already existed in the 2LT framework, we extended it to work for spreadsheet models. Assuming that the column `"Tax tariff"` was already inserted, we can run the following functions:

```
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)
 ...
```

Note that above result 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 and bottom part on the top and 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, $normalize$, explained in Section 4.3.

Make it Expandable: It is possible to make a block in a class expandable. For this, we created the 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)↓))
```

It receives the label of the class to make expandable and updates the class to allow repetition. The result type constructor is $\cdot : (\cdot)^{\downarrow}$; the to function wraps the existing

block into a list, *tolist*; and the *from* function takes the head of it, *head*. We developed a similar rule to make a class expandable. This corresponds to promote a class c to $c^{\rightarrow}$. We do not show its implementation here since it is quite similar to this one.

Split: It is quite common to move a column in a spreadsheet from on place to another. The rule *split* copies a column to another place and substitutes the original column values by references to the new column (similar to create a pointer). The rule to move part of the spreadsheet is presented in Section 4.3. The first step of *split* is to get the column that we want to copy:

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

If the corresponding label is found, the vertical composition is returned. Note that, as in other rules, this rule is intended to be applied using the combinator *once*. As we said, we aim to write local rules that can be used at any level using the developed combinators.

The rule creates in a second step 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 original column that was copied into references to the new column. The rule $makeReferences :: String \rightarrow Rule$ receives the label of the column that was copied (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 (with the help of *once*) to it we get the following new model:

```
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))↓)
```

4.3 Layout Rules

In this section we describe rules focused on the layout of spreadsheets, that is, rules that do not add/remove information to/from the model.

Change Orientation: The rule *toVertical* changes the orientation of a block from horizontal to vertical.

```
toVertical :: Rule
toVertical (a ⫞ b) = return (View idrep (a^b))
```

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.

Normalize Blocks: When applying some transformations, the resulting types may not have the correct shape. A common example is to have as result the following type:

```
A ⫞ B^C ⫞ D^
E ⫞ F
```

Most of the times, the correct result is the following:

```
A ⫞ B ⫞ D^
E ⫞ C ⫞ 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_1 and π_2.

```
normalize1 :: Rule
normalize1 (a ⫞ b^c ⫞ d^e ⫞ f) =
  let tof = id × π1 × id ∘ π1△π1 ∘ π2△π2 ∘ π1 ∘ π2 × π2
      fromf = π1 ∘ π1△π1 ∘ π2 × π1 ∘ π2△π2 ∘ π2 ∘ π1△id × π2 ∘ π2
  return (View (Rep {to = tof, from = fromf}) (a ⫞ b ⫞ d^e ⫞ c ⫞ f))
```

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

Shift: It is quite common to move parts of the spreadsheet across it. We designed a rule to shift parts 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
  let rep = Rep {to = pnt (⊥ :: EmptyBlock)△id, from = π2}
  return (View rep (EmptyBlock ⫞ b1))
```

The function *teq* verifies if two types are equal. This rule receives a type and a block, but we can easily write a wrapper function to receive a label in the same style of *insertCol*.

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) | isBlock 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 projections 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.

Move Blocks: A more complex task 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)
```

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'}
```

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

5 Related Work

Ko *et al.* [13] summarize and classify the research challenges of the end user software engineering area. These include requirements gathering, design, specification, reuse, testing and debugging. However, besides the importance of Lehman's laws of software evolution [14], 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.* [15] 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. They present a set of rules and a prototype tool to support these changes. In this paper we present a more advanced way to evolve spreadsheets models and instances in a different way: first, we use strategic programming with two-level coupled transformation. 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, that is, it allows the migration of data from the new model back to the old model.

Vermolen and Visser [23] proposed 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 allows 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 their approach is tailored for forward evolution, our approach supports reverse engineering, that is, it supports automatic transformation and migration from the new model to the old model.

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 capturing 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 GADT with embedded point-free function representations. This model is reminiscent of the ClassSheet.
- 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. This system combines strategy combinators known from strategic programming with spreadsheet-specific transformation rules.
- We have illustrated our approach with a number of specific spreadsheet refactorings to perform the evolution of spreadsheets.

The rules here presented are implemented in the HAEXCEL framework consisting of a set of libraries providing functionality to load (from different formats), transform, infer spreadsheet models (e.g. ClassSheet), and, now, perform the co-evolution of such models their (spreadsheet) instances. HAEXCEL includes an add-on for OpenOffice. Currently, we are integrating the rules presented in this paper, in a spreadsheet programming environment where end users can interact both with the model and the data [9].

References

1. Abraham, R., Erwig, M.: Inferring templates from spreadsheets. In: Proc. of the 28th Int. Conference on Software Engineering, pp. 182–191. ACM, New York (2006)
2. Abraham, R., Erwig, M., Kollmansberger, S., Seifert, E.: Visual specifications of correct spreadsheets. In: Proc. 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.N., Visser, J.: Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 284–299. Springer, Heidelberg (2006)
5. Cunha, A., Visser, J.: Strongly typed rewriting for coupled software transformation. Electronic Notes on Theoretical Computer Science 174, 17–34 (2007)
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, pp. 11–20. ACM, New York (2007)
7. Cunha, J., Beckwith, L., Fernandes, J.P., Saraiva, J.: An empirical study on the influence of different spreadsheet models on end-users performance. Tech. Rep. DI-CCTC-10-10, CCTC, Departamento de Informática, Universidade do Minho (2010)

8. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: Proc. of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 93–100. IEEE Computer Society, Washington, DC, USA (2010)
9. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Embedding spreadsheet models in spreadsheet systems (2011) (submitted for publication)
10. Cunha, J., Saraiva, J., Visser, J.: Discovery-based edit assistance for spreadsheets. In: Proc. of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 233–237. IEEE Computer Society, Washington, DC, USA (2009)
11. Engels, G., Erwig, M.: ClassSheets: Automatic generation of spreadsheet applications from object-oriented specifications. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 124–133. ACM, New York (2005)
12. Hinze, R., Löh, A., Oliveira, B.: "Scrap Your Boilerplate" Reloaded. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 13–29. Springer, Heidelberg (2006)
13. 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. J. ACM Computing Surveys (2009)
14. Lehman, M.M.: Laws of software evolution revisited. In: Montangero, C. (ed.) EWSPT 1996. LNCS, vol. 1149, pp. 108–124. Springer, Heidelberg (1996)
15. Luckey, M., Erwig, M., Engels, G.: Systematic evolution of typed (model-based) spreadsheet applications (submitted for publication)
16. Morgan, C., Gardiner, P.H.B.: Data refinement by calculation. Acta Informatica 27, 481–503 (1990)
17. Oliveira, J.N.: A reification calculus for model-oriented software specification. Formal Aspects of Computing 2(1), 1–23 (1990)
18. Oliveira, J.N.: Transforming data by calculation. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 134–195. Springer, Heidelberg (2008)
19. Panko, R.R.: Spreadsheet errors: What we know. What we think we can do. In: Proceedings of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (July 2000)
20. Peyton Jones, S., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types. Tech. Rep. MS-CIS-05-26, Univ. of Pennsylvania (July 2004)
21. Powell, S.G., Baker, K.R.: The Art of Modeling with Spreadsheets. John Wiley & Sons, Inc., New York (2003)
22. Rajalingham, K., Chadwick, D., Knight, B.: Classification of spreadsheet errors. In: European Spreadsheet Risks Interest Group, EuSpRIG (2001)
23. Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 630–644. Springer, Heidelberg (2008)