JFP **16** (3): 293–325, 2006. © 2005 Cambridge University Press
doi:10.1017/S0956796805005794 First published online 16 November 2005 Printed in the United Kingdom

Gencel: a program generator for correct spreadsheets[*]

MARTIN ERWIG, ROBIN ABRAHAM, STEVE KOLLMANSBERGER
School of EECS, Oregon State University, OR, USA
(*e-mail:* {erwig,abraharo,kollmast}@eecs.oregonstate.edu)

IRENE COOPERSTEIN[†]
Department of CS, University of Houston, Houston, TX, USA
(*e-mail:* Irene.Cooperstein@mail.uh.edu)

Abstract

A huge discrepancy between theory and practice exists in one popular application area of functional programming – spreadsheets. Although spreadsheets are the most frequently used (functional) programs, they fall short of the quality level that is expected of functional programs, which is evidenced by the fact that existing spreadsheets contain many errors, some of which have serious impacts. We have developed a template specification language that allows the definition of spreadsheet templates that describe possible spreadsheet evolutions. This language is based on a table calculus that formally captures the process of creating and modifying spreadsheets. We have developed a type system for this calculus that can prevent type, reference, and omission errors from occurring in spreadsheets. On the basis of the table calculus we have developed Gencel, a system for generating reliable spreadsheets. We have implemented a prototype version of Gencel as an extension of Excel.

1 Introduction

Spreadsheets are functional programs (Peyton Jones *et al.*, 2003). Although spreadsheets offer only a subset of the functionality of modern functional programming languages, they are widely used: It is estimated that each year tens of millions of professionals and managers create hundreds of millions of spreadsheets (Panko, 2000). These numbers indicate not only that spreadsheet systems are among the most frequently used software systems, they also show that spreadsheets are the most frequently employed functional programs. This also means that functional programs outnumber by far all other programs in all other programming paradigms. What a success of functional programming!

Not quite. One of the distinguishing claims of functional programming is that functional programs are more reliable than, for example, imperative programs, and contain fewer errors. To some degree the increased reliability is achieved

[*] This work is partially supported by the National Science Foundation under the grant ITR-0325273 and by the EUSES Consortium (http://EUSESconsortium.org).
[†] The work of Irene Cooperstein was performed at Oregon State University during summer 2004 as part of the CRA-W Distributed Mentor Project.

through a cleaner language design and through sophisticated type systems that help to detect program errors early. Unfortunately, spreadsheets suffer heavily from errors. Numerous studies have shown that existing spreadsheets contain errors at an alarmingly high rate (Brown & Gould, 1987; Lerch *et al.*, 1989; Panko, 2000). Some studies even report that 90% or more of real-world spreadsheets contain errors (Rajalingham *et al.*, 2001). This situation should not be too surprising given the facts that (a) spreadsheet systems offer only weak or no typing at all and (b) the language in which spreadsheets are "written" is seldom given in an explicit form with well-defined syntax and semantics. Instead the language is specific to a particular spreadsheet system. (In other words, a spreadsheet system is essentially an IDE for a particular spreadsheet language that is implicitly defined through the features of the spreadsheet system.)

Imagine if we could bring some of the advantages of functional programming with respect to safety and reliability to the realm of spreadsheets. This would have a great impact in two major respects: First, it would make spreadsheet programs more reliable. Second, it would boost the attention level for functional programming. Altogether, this would mean a big success for functional programming in the real world.

Why has programming language research not taken spreadsheets seriously? One reason might be that spreadsheets are considered to be trivial and not sufficiently challenging. After all, spreadsheets are just simple first-order, non-recursive programs with non-nested bindings, so why bother at all? Although this characterization is accurate, the comparison is based on a static view of one particular spreadsheet and ignores update operations in spreadsheets. However, much of the success of spreadsheet systems is due to their interactive nature allowing changes to input data and the spreadsheet program with immediate feedback after changes have been performed (Hendry & Green, 1994; Lewis & Olson, 1987; Norman, 1986; Kay, 1984). Unfortunately, many spreadsheet errors are a consequence of how this form of interaction is realized.

One major problem in existing spreadsheet systems is that the same user-interface actions are used to change a program *and* its input. For example, placing a number in a cell that already contains a (different) number means to change an input argument of the spreadsheet, which causes a new run of the spreadsheet program. In contrast, placing a number in a cell that contains a formula changes the spreadsheet program, which also causes the immediate rerunning of the spreadsheet program. This overloading can lead to the introduction of errors through unintended overriding of formulas. Other sources of errors are the inconsistent definition of insert-row and insert-column commands, which trigger the automatic adjustment of ranges in aggregation formulas only in some cases. These errors are particularly insidious since in many cases they creep into a spreadsheet unnoticed.

Since update operations are a major source of spreadsheet errors, we propose to specify the possible evolutions of a spreadsheet in advance and to create customized update operations for any such specification. The benefit of this approach is that users still can apply update operations to their spreadsheets, but only those that keep the spreadsheet within the specified evolution and that do not introduce any reference, type, or omission errors.

In the following we will present the spreadsheet system Gencel, which has been implemented as an extension of Excel. A preliminary description of the system is given in Erwig *et al.* (2005). Our approach to improving the reliability of spreadsheets is to ensure the correctness not just of a single spreadsheet, but of all the spreadsheets into which it can evolve over time. To this end we have defined a specification language to describe spreadsheets and their possible evolutions through *templates*. Any such template is translated into an initial spreadsheet and a set of spreadsheet update operations that are tailored to this particular spreadsheet and ensure that the spreadsheet can be changed only according to the template. Moreover, we have defined a type system for the specification language that can guarantee the following form of *spreadsheet evolution correctness*: Any spreadsheet that evolves from a well-typed template will not contain any reference or type errors.

In section 2 we illustrate the idea of using program generation to support the creation of safely evolvable spreadsheet. In section 3 we define syntax and semantics of a table calculus that forms the formal foundation of our Gencel system. A type system for the table calculus is developed in section 4 to guarantee that well typed templates will be transformed into customized spreadsheets that can evolve only without errors. This safety result is presented in section 5. In section 6 we describe the implementation of the Gencel system as an extension of Excel. Related work is discussed in section 7. We present conclusions and directions for future research in section 8.

2 A spreadsheet generator

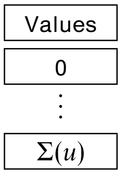
We regard a spreadsheet as a collection of tables where a table is essentially a rectangular area consisting of a number of rows and columns. Changes in one table of a spreadsheet should not affect other tables unless they contain a reference to the changed table. For simplicity, we ignore the possibility of references between different tables in this paper and therefore consider in the following only the specification of single tables.

The tables in a spreadsheet often change over time. However, at any given time only a subset of all possible changes to a table are reasonable. The decision whether a particular update should be allowed or prohibited depends on the roles of the affected cells in the table. From the point of view of a spreadsheet application, the cells of a table can be distinguished into label, data, and computation cells. Moreover, some rows or columns of a table are fixed, like header and footer rows and columns, while other rows and columns are duplicated if new data is to be added.

The template specification language that is part of our Gencel system reflects this view and offers constructs to define a template as a horizontal sequence of fixed and extendable columns where a column is constructed as a vertical sequence of fixed and extendable blocks, which are rectangular collections of cells containing values and formulas. Note that the alternative view as a sequence of rows is also possible. However, allowing this alternative representation would not add any functionality. Therefore, we have fixed the representation to simplify the formal model.

In the following we will illustrate the idea of table generation through several examples. The templates will be given in a visual notation called ViTSL (an acronym for **Vi**sual **T**emplate **S**pecification **L**anguage). A corresponding textual representation will be presented in Section 3 where we introduce the table evolution calculus.

Our first example is the specification of a plain column of numbers with a header at the top and a summation formula at the bottom. This template can be specified by the following ViTSL expression.

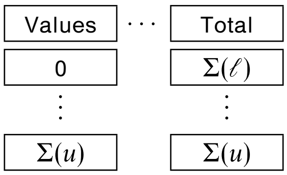


The template consists of three elements: the header containing the label, the footer containing the summation formula, and a vertically expandable group (also called *vex group* for short) that consists of a single cell containing the value 0. The argument of the summation formula, u, is a relative reference to the vex group above it. (Note that u means "up" and is simply a name for $(0,-1)$.)

The template describes a class of tables that all consist of one column with the shown header and footer and that have one or more number cells in between. This template can be compiled into an initial Excel spreadsheet together with customized definitions for all spreadsheet update operations that ensure that only tables matching the template will be created. ViTSL offers the following visual elements for templates:

- *Cells*, represented by rectangles and containing formulas.
- *References*, represented by (names for) relative grid offsets.
- *Vex groups*, represented by vertical dots that indicate the possible expansion of one or more cells in the vertical direction.
- *Hex groups*, represented by horizontal dots that indicate the possible expansion of one or more columns in the horizontal direction.

An example of a horizontally expandable group (*hex group*) is given in the following template for a summation table:

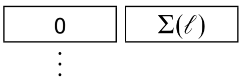


Here the summation column from the previous example is horizontally expandable and is horizontally joined by a column that also contains a header and a summation footer, but whose vex group contains a summation formula whose argument references the number cell of the hex group.

The hex group in the last example illustrates that expandable groups may consist of groups of cells and not just single cells. Moreover, one column can also contain multiple vex groups. Similarly, a template can contain multiple hex groups. However, vex groups and hex groups cannot be arbitrarily nested. The only possible nesting is indicated by the example: Hex groups may contain vex groups. There are two aspects of this restriction. First, it prevents nested expansion groups in one dimension, which is important to keep the spreadsheet user interface simple, because to be able to work with nested expansion groups in one dimension a more sophisticated notion of "position" would be needed: Just knowing, for example, the current row number is not enough to tell whether to insert a new row in the innermost expansion group or a new group in the enclosing one. The second aspect is the restriction that vex groups cannot contain hex groups, which is just for technical reasons to keep the formalism simple. Since columns have to be vertically aligned, templates in which hex groups are nested in vex groups would not add to the expressiveness of the language and could be transformed into a corresponding template in which vex groups are nested in hex groups.

In addition, several structural constraints are needed to ensure that a reasonable definition for the spreadsheet update operations exists. For example, all columns in a template have to *align* vertically. To explain the idea of alignment, consider a column as a sequence of non-vex and vex groups, say $c = [b_1, \ldots, b_k]$. Now c matches another column $c' = [b'_1, \ldots, b'_k]$ only if (a) b_i has the same height as b'_i and (b) b_i is an expandable group iff b'_i is. These constraints ensure, in particular, that vex groups are horizontally aligned and have the same height, which allows the insert-row command to be defined to insert a number of rows according to the common height of the vex groups. Similarly, we require that all blocks in a column have the same width. For columns in hex groups, this constraint ensures that the insert-column command can be defined to create a number of columns according to the common width of the blocks of the hex group.

We can consider an example that violates these constraints to see why they are required. Consider the case for the following template:



In this template, we have the horizontal composition of two columns – the left column repeats vertically whereas the right column is simply a reference. These columns do not align and thus the above template is not legal. In the initial table, we would have one value with a single reference to that value, which causes no problems. However, if we consider the insertion of rows, we can observe that additional values would be added. What would be the meaning of the reference in that case? Would it refer merely to the topmost value or to all of the values? In addition, the vertical and horizontal concatenation of blocks assumes that, at all times, blocks, columns and tables will be rectangular. However, we can see a non-rectangular shape emerging in this example. How does such a shape concatenate with other columns or tables? These uncertainties have led us to forbid such templates.

An example for a template containing multiple vex groups is the accounting sheet shown below.

Income
0
⋮
$\Sigma(u)$
Expenses
0
⋮
$\Sigma(u)$
Net Earnings
$\Delta(u^5, u^2)$

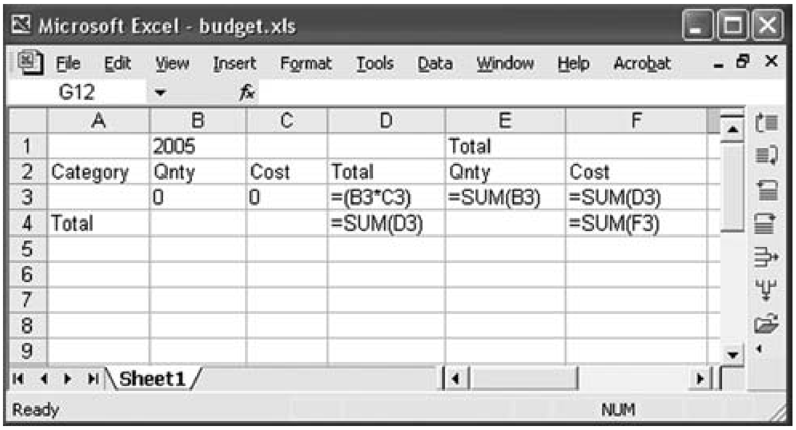
The gaps between cells indicate the scope of the vertical dots. For example, the first vex group is the repeated 0 and not the block of the top two cells. Similarly, the second vex group consists only of the one cell containing the 0, which is below the cell labeled Expenses. The formula $\Delta(u^5, u^2)$ computes the difference between the two summation cells. An exponent k attached to a relative reference means the k-fold repetition of the reference. For example, u^2 refers to two cells above. Since the vertical (and horizontal) dots are not cells on their own, they are not counted when determining relative references. Therefore, u^5 refers to the upper vex group and u^2 to the lower one.

The relative references used in templates are very expressive: First, unlike absolute addresses, relative references are compositional, that is, they need not be adjusted when cells or blocks are composed with other blocks. Second, depending on their origin and target, relative references can express single-cell addresses as well as ranges. For example, the references from the summation formulas point into a vex group and refer to all the cells that will be generated within that group, in contrast to the references u^5 and u^2, which point to nonexpandable cells and refer always to single values.

As a final example we present a template for a multi-year budgeting sheet that contains a multi-column hex group.

	2005			⋯	Total	
Category	Qnty	Cost	Total		Qnty	Cost
	0	0	$\Pi(\ell^2, \ell)$		$\Sigma(\ell^3)$	$\Sigma(\ell^2)$
⋮		⋮			⋮	
Total			$\Sigma(u)$			$\Sigma(u)$

The multi-column hex group illustrates another powerful feature of our model that goes beyond Excel's update capabilities – the possibility of automatically maintaining *non-consecutive ranges* over insert and delete operations. Consider, for



Microsoft Excel - budget.xls

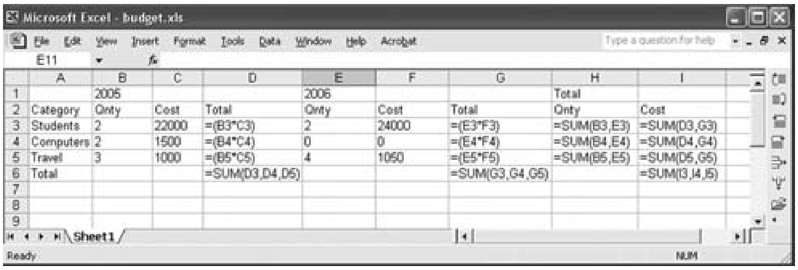
G12

	A	B	C	D	E	F
1		2005			Total	
2	Category	Qnty	Cost	Total	Qnty	Cost
3		0	0	=(B3*C3)	=SUM(B3)	=SUM(D3)
4	Total			=SUM(D3)		=SUM(F3)
5						
6						
7						
8						
9						

Sheet1

Ready NUM

Fig. 1. Generated Excel budget spreadsheet.



Microsoft Excel - budget.xls

E11

	A	B	C	D	E	F	G	H	I
1		2005			2006			Total	
2	Category	Qnty	Cost	Total	Qnty	Cost	Total	Qnty	Cost
3	Students	2	22000	=(B3*C3)	2	24000	=(E3*F3)	=SUM(B3,E3)	=SUM(D3,G3)
4	Computers	2	1500	=(B4*C4)	0	0	=(E4*F4)	=SUM(B4,E4)	=SUM(D4,G4)
5	Travel	3	1000	=(B5*C5)	4	1060	=(E5*F5)	=SUM(B5,E5)	=SUM(D5,G5)
6	Total			=SUM(D3,D4,D5)			=SUM(G3,G4,G5)		=SUM(I3,I4,I5)
7									
8									
9									

Sheet1

Ready NUM

Fig. 2. Updated Excel budget spreadsheet.

example, the rightmost summation formula $\Sigma(\ell^2)$. The relative reference refers to the cell containing the Π formula, which represents a product. The repeated application of insert-column commands generates several non-adjacent instances of that cell. Nevertheless, the update operations created by the Gencel system will properly update the rightmost summation formula to always contain references to exactly all those cells (see Figure 2). The same is true for the Qnty summation formula.

The initial spreadsheet generated from the budget-sheet specification is shown in Figure 1.

After one column and two row insertions and several changes to the stored values, the spreadsheet might look as shown in Figure 2. Note, in particular, how the ranges in the SUM formulas in rows H and I represent non-consecutive ranges.

The Gencel system offers additional buttons for inserting columns to the left and right of the current position as well as for inserting rows above and below the current row. Note that it is not possible to enter values of wrong types or to change or delete existing formulas.

For illustration we show here the formula view of the generated spreadsheet. End users will generally only see the computed values. The spreadsheet in Figure 2 is created by precisely following the formal definitions of the table calculus. We can optimize the generated formulas further by compressing ranges, which yields, for example, SUM(D3:D5) in cell D6.

3 The table evolution calculus

The *table evolution calculus* provides a formal foundation for the Gencel system. In section 3.1 we will define its syntax. In section 3.2 we define the semantics, which consists of the generation of tables from templates, the definition of table update operations, which define the possible evolutions of tables, and the reduction of tables into tables containing just values.

3.1 Syntax

A *template* (t) is given by a horizontal composition ($\shortmid$) of fixed (c) or expandable ($c^{\rightarrow}$) columns, where a column is given by a vertical composition ($\hat{}$) of fixed (b) or expandable ($b^{\downarrow}$) blocks. A block is given by a composition of formulas (f). Blocks are also used to represent plain *tables*. Formulas consist of basic values (ϕ), references (ρ), and expressions that are built by applying functions to a varying number of arguments given by formulas ($\phi(f,\ldots,f)$). In this simple version of the table calculus we only use functions that can be applied to an arbitrary number of arguments of the same type, like addition (Σ) and multiplication (Π). This restriction simplifies the semantics of formulas and the type system a bit, but is not essential.

References are given by pairs of integers and represent relative references in the form of offsets, that is, a reference (v,h) means to go v cells to the right and h cells up. We use the following abbreviations for cell offsets: $\ell = (-1,0)$, $r = (1,0)$, $u = (0,-1)$, and $d = (0,1)$. We sometimes use sequences of abbreviated offsets to represent larger offsets, for example, $\ell\ell = \ell^2 = (-2,0)$.

The syntax of templates is summarized in Figure 3.

$$
\begin{array}{llll}
f \in \mathit{Fml} & ::= & \phi \mid \rho \mid \phi(f,\ldots,f) & (\mathit{formulas}) \\
b \in \mathit{Block} & ::= & f \mid b \shortmid b \mid b \mathbin{\hat{}} b & (\mathit{blocks,\ tables}) \\
c \in \mathit{Col} & ::= & b \mid b^{\downarrow} \mid c \mathbin{\hat{}} c & (\mathit{columns}) \\
t \in \mathit{Template} & ::= & c \mid c^{\rightarrow} \mid t \shortmid t & (\mathit{templates})
\end{array}
$$

Fig. 3. Templates.

The constructs correspond directly to the visual notation. Whenever we want to talk about an arbitrary repeating group, that is, either a vex or a hex group, we also use the notation u^{+} where the metavariable u ranges over columns and blocks. We also define that $\hat{}$ and $\shortmid$ associate to the left.

As an example, consider the summation column, which was shown as the first ViTSL example in section 2. This column is represented by the following template.

$$\mathsf{Values} \mathbin{\hat{}} 0^{\downarrow} \mathbin{\hat{}} \Sigma(u)$$

We refer to this expression as *SumCol* in the following. The summation table is represented by the following template, which we name *SumTab*.

$$(\mathsf{Values} \mathbin{\hat{}} 0^{\downarrow} \mathbin{\hat{}} \Sigma(u))^{\rightarrow} \shortmid \mathsf{Total} \mathbin{\hat{}} \Sigma(\ell)^{\downarrow} \mathbin{\hat{}} \Sigma(u)$$

We introduce as a structure to support the semantics definition a generalization of the concept of template in which we represent the number of expansions for each

Table 1. *Structures used in the semantics of Gencel*

Structure/Concept	contains ...
template (t)	$\hat{}$, $\mid$, $b^{\downarrow}$, $c^{\rightarrow}$, and u^{+}
template instance ($\underline{t}$)	$\hat{}$, $\mid$, $b^{\mid k}$, $c^{\underline{k}}$, and u^{k}
table (= *block*) (b)	$\hat{}$ and $\mid$
repetition (k)	exponent in $b^{\mid k}$, $c^{\underline{k}}$, or u^{k}
expansion area (b)	subpart of a table

vex and hex group. This structure is called *template instance*; its syntax is identical to the syntax of templates in Figure 3 except that $b^{\downarrow}$ and $c^{\rightarrow}$ are replaced by $b^{\mid k}$ and $c^{\underline{k}}$, respectively. We use the metavariable $\underline{t}$ to range over template instances. Similar to repeating groups in templates, we use the abbreviation u^k to represent an arbitrary vex or hex group in a template instance. A column c (from a template) of width w that is expanded k times in a template instance corresponds in the generated table to kw columns. This whole area in the table is called c's *expansion area*, and k is called c's *repetition*. Likewise, a block b of height h that is expanded k times corresponds in the generated table to a rectangular area of height hk (and width of b). Again, this area is called b's *expansion area*, and k is called b's *repetition*.

We summarize all structures/concepts and their distinguishing characteristics explicitly in Table 1 for easy future reference.

3.2 Semantics

The semantics of the table calculus consists of three parts: (1) the translation of templates into initial tables (sections 3.2.1 through 3.2.3), (2) the semantics of table update operations relative to a template (section 3.2.4), and (3) the evaluation of tables (section 3.2.5). In the following we will describe all these steps in some detail.

3.2.1 Generating template instances

The function $\mathscr{I}$ produces a template instance from a template by simply replacing each "$\rightarrow$" or "$\downarrow$" exponent by a fixed exponent n. For the purpose of spreadsheet generation, we need to use $\mathscr{I}$ only with the index 1, but in the definition of the type system in Section 4 we will use it with a different exponent to identify cells in repeating groups. We use the metavariables u and v to range over *Template*, *Col*, and *Block*, which allows us to give some definitions more concisely. For example, in the definition for $\mathscr{I}$, we can combine the cases for horizontal and vertical repetition.

$$
\begin{aligned}
\mathscr{I}_n(t \mid t') &= \mathscr{I}_n(t) \mid \mathscr{I}_n(t') \\
\mathscr{I}_n(c \hat{} c') &= \mathscr{I}_n(c) \hat{} \mathscr{I}_n(c') \\
\mathscr{I}_n(u^{+}) &= (\mathscr{I}_n(u))^n \\
\mathscr{I}_n(b) &= b
\end{aligned}
$$

We employ the following auxiliary functions for computing the width and height of templates.[1]

$$
\begin{array}{ll@{\qquad\qquad}ll}
\overleftrightarrow{f} & = 1 & \updownarrow f & = 1 \\
\overleftrightarrow{u \mid v} & = \overleftrightarrow{u} + \overleftrightarrow{v} & \updownarrow u \mid v & = \max(\updownarrow u, \updownarrow v) \\
\overleftrightarrow{u \hat{\ } v} & = \max(\overleftrightarrow{u}, \overleftrightarrow{v}) & \updownarrow u \hat{\ } v & = \updownarrow u + \updownarrow v \\
\overleftrightarrow{u^{+}} & = \overleftrightarrow{u} & \updownarrow u^{+} & = \updownarrow u \\
\overleftrightarrow{b^{|k}} & = \overleftrightarrow{b} & \updownarrow b^{|k} & = k \updownarrow b \\
\overleftrightarrow{c^{\underline{k}}} & = k\overleftrightarrow{c} & \updownarrow c^{\underline{k}} & = \updownarrow c
\end{array}
$$

The following function locates cells in templates, template instances, and tables based on absolute references.

$$
\begin{array}{ll}
f[1,1] & = f \\
(u \mid v)[x,y] & = \begin{cases} u[x,y] & \text{if } x \leqslant \overleftrightarrow{u} \\ v[x - \overleftrightarrow{u}, y] & \text{otherwise} \end{cases} \\
(u \hat{\ } v)[x,y] & = \begin{cases} u[x,y] & \text{if } y \leqslant \updownarrow u \\ v[x, y - \updownarrow u] & \text{otherwise} \end{cases} \\
u^{+}[\rho] & = u[\rho] \\
b^{|k}[x,y] & = \begin{cases} b[x, ((y-1) \bmod \updownarrow b) + 1] & \text{if } y \leqslant k \updownarrow b \\ \perp & \text{otherwise} \end{cases} \\
c^{\underline{k}}[x,y] & = \begin{cases} c[((x-1) \bmod \overleftrightarrow{c}) + 1, y] & \text{if } x \leqslant k\overleftrightarrow{c} \\ \perp & \text{otherwise} \end{cases}
\end{array}
$$

The last two cases allow applications of the lookup function to work on template instances. In section 4 we will use the function also on template types.

Next we define the function $\mathscr{G}$ for generating a table from a template. In fact, we define a slightly more general function that works on template instances and that can be reused in the definition of the update operations. In the initial table all $\rightarrow$ and $\downarrow$ exponents are replaced by ones. Then each application of an insert-column command increases the exponent of one hex group by one, whereas each application of an insert-row command increases the exponents of all vex groups in one row by one.

A template instance contains sufficient information to (re)generate all formulas with all correct references for the corresponding table. This fact is exploited in the definition of the update operations, which essentially create an updated template instance and derive the changed formulas from the new instance. Template instances do not contain the current values from the actual table, which is not problematic

[1] Since we will reuse these auxiliary functions in the definition of the type system, we define them more generally for arbitrary templates and template instances.

since they are not needed for the definition of the update operations, because all newly inserted values are taken from the template.

3.2.2 Translation of relative references

We have to pay particular attention to the generation of absolute addresses from relative references in formulas. In particular, when a relative reference points into a vex or hex group, the reference might mean a range of cells that have been expanded from the cell in the specification. However, this is only the case if the relative reference points to a vex or hex group that is not in the same horizontal or vertical expansion area, that is, an insert-row or insert-column command in that vex/hex group does not cause a duplication of the cell containing the reference as well, because in that case the reference would be just a reference to a "parallel" developing expandable block. In other words, the generation of absolute references has to distinguish whether or not the referenced cell b is *expand-dependent* on the cell a that contains the reference. A cell b is expand-dependent on a cell a if b is expanded only if a is also expanded. In particular, this is true whenever b is not expandable at all.

We can convert a relative reference $\rho = (i, j)$ contained in a cell $a = (x, y)$ into an absolute reference with four steps. First, we determine a single absolute address by adding the relative reference to the cell address, giving $b = (x + i, y + j)$. Second, we determine the range of cells for the expansion area of the target address b. Third, to account for expansion dependency (that is, when the source and target cell expand in an aligned way), we remove from the calculated range the set of cells whose x or y coordinate falls within the expansion area of the source address a. The only exception to this removal is the block containing a, since it is possible that b is within this block. Finally, we ensure that the source cell a is not in the referenced set, while the original target cell b is. In the following we formalize these four steps.

First, we define an auxiliary function H to compute for a template instance the horizontal range of coordinates that is covered by the hex group containing a particular address. Such a range is represented by a triple (x_1, x_2, w) where x_1 and x_2 describe the vertical range and w gives the width of the hex group. In the following definitions we use $\oplus$ to add a value to the first two components of a range triple, that is, $i \oplus (x_1, x_2, w) = (i + x_1, i + x_2, w)$.

$$
\begin{aligned}
H^x(f) &= (x, x, 1) \\
H^x(u \mid v) &= \begin{cases} \overleftrightarrow{u} \oplus H^{x-\overleftrightarrow{u}}(v) & \text{if } x > \overleftrightarrow{u} \\ H^x(u) & \text{otherwise} \end{cases} \\
H^x(u \hat{\ } v) &= H^x(u) \\
H^x(b^{|k}) &= H^x(b) \\
H^x(c^{\underline{k}}) &= \begin{cases} (1, k\overleftrightarrow{c}, \overleftrightarrow{c}) & \text{if } x \leqslant k\overleftrightarrow{c} \\ \perp & \text{otherwise} \end{cases}
\end{aligned}
$$

Similarly, V computes the vertical range (y_1, y_2, h) of the vex group covering a cell where h represents the height of the vex group.

$$\begin{aligned}
V_y(f) &= (y, y, 1)\\
V_y(u \mid v) &= V_y(u)\\
V_y(u\,\hat{}\,v) &= \begin{cases} \updownarrow u \oplus V_{y-\updownarrow u}(v) & \text{if } y > \updownarrow u\\ V_y(u) & \text{otherwise}\end{cases}\\
V_y(b^{|k}) &= \begin{cases} (1, k\updownarrow b, \updownarrow b) & \text{if } y \leqslant k\updownarrow b\\ \bot & \text{otherwise}\end{cases}\\
V_y(c^{\underline{k}}) &= V_y(c)
\end{aligned}$$

As an illustration of how the functions H and V work consider the following instance of the template *SumTab*.

$$\underline{t} = (\mathsf{Values}\,\hat{}\,0^{|3}\,\hat{}\,\Sigma(u))^{\underline{2}} \mid \mathsf{Totals}\,\hat{}\,\Sigma(\ell)^{|3}\,\hat{}\,\Sigma(u)$$

Let us examine the cell at absolute address $(3, 2)$ using the H and V functions. Initially, we break up $\underline{t}$ into $u = (\mathsf{Values}\,\hat{}\,0^{|3}\,\hat{}\,\Sigma(u))^{\underline{2}}$ and $v = \mathsf{Totals}\,\hat{}\,\Sigma(\ell)^{|3}\,\hat{}\,\Sigma(u)$, and we apply the second rule of H, that is, $H^3(u \mid v) = 2 \oplus H^1(v)$. The region of interest lies within v, the right-most horizontal position. Examining v, we see that it is constructed from three vertically composed blocks: $\mathsf{Totals}\,\hat{}\,\Sigma(\ell)^{|3}$, represented by u', and $\Sigma(u)$, represented by v'. This structure matches the third rule of H, that is, $H^1(u'\,\hat{}\,v') = H^1(u')$. Because we are computing only horizontal data, these vertically composed blocks are insignificant. However, u' is still formed from two vertically composed blocks, so we repeat this step once more, which leads to $H^1(u''\,\hat{}\,v'') = H^1(u'')$. Now, since u'' is merely the value Totals, the expression matches the final rule of H, and we obtain $H^1(u'') = (1, 1, 1)$. The final output of the function H is thus $H^3(\underline{t}) = 2 \oplus (1, 1, 1) = (3, 3, 1)$. The first value indicates that the region containing the cell at absolute address $(3, 2)$ begins at the horizontal coordinate 3. The second value indicates that the region also ends at the horizontal coordinate 3 – in other words, it consists of only a single column. The third value denotes the width of the region, which, as we knew already from the previous data, is merely 1.

Next we examine $V_2(u \mid v)$ to determine the vertical range. We obtain $V_2(u \mid v) = V_2(u)$, which indicates the insignificance of horizontal composition to the V function. We now have $u = u'\,\hat{}\,v'$, where u' is $\mathsf{Values}\,\hat{}\,0^{|3}$ and v' is $\Sigma(u)$. We have $V_2(u'\,\hat{}\,v') = V_2(u')$. Since the height of u' is 4, this step concludes that the vertical region of interest is within u'. Now break up u' further into $u'' = \mathsf{Values}$ and $v'' = 0^{|3}$. We obtain $V_2(u''\,\hat{}\,v'') = 1 \oplus V_1(v'')$. Finally, we obtain $V_1(0^{|3}) = (1, 3\updownarrow 0, \updownarrow 0) = (1, 3, 1)$. The final result is therefore $V_2(\underline{t}) = 1 \oplus (1, 3, 1) = (2, 4, 1)$. The first value indicates that the vertical region begins at y coordinate 2 and ends at y coordinate 4 (making it a total of 3 cells tall); the height of a single repeating block is 1.

With the help of H and V we can define a function T that translates relative offsets into absolute "target" addresses. If a is not a cell that has been expanded from a repeating group or if $a + \rho$ does not leave the repeating group containing a, b is obtained by simply adding ρ to a. Otherwise, b is obtained by adding ρ to the first generated cell from a. For example, in all instances of the following template the reference ℓ^2 should refer to the cell $(1, 1)$ that contains the 0.

$$0 \mid (9 \mid \ell^2)^{\rightarrow}$$

For the initial template instance $0 \mid (9 \mid \ell^2)^{\underline{1}}$, $(1, 1)$ is indeed obtained by simply adding the relative reference, that is, $(3, 1) + \ell^2 = (3, 1) + (-2, 0) = (1, 1)$. However, this is not the case for the instance $0 \mid (9 \mid \ell^2)^{\underline{2}}$, in which ℓ^2 in the rightmost cell simply points to $(3, 1)$. In this case, adding ℓ^2 to the first of the generated cells, which has the address $(3, 1)$, works. On the other hand, the reference ℓ in the template $0 \mid (9 \mid \ell)^{\rightarrow}$ always refers to the cell directly left to it, which contains the 9. The situation is analogous for vex groups.

This distinction is reflected in the definition of the function T as follows. First, the first instance of (x, y) in the repeating group is computed by determining the offset that x and y have from the start of the repeating block (given by $(x - x_1) \bmod w$ and $(y - y_1) \bmod h$) and adding this offset to the first cell of the whole range, which is (x_1, y_1). Second, we determine a tentative "target cell" (x_t, y_t) by adding ρ to the first instance. For (x_t, y_t) we then check for each dimension whether or not it is contained in the corresponding range of (x, y), which is given by (x_1, x_2) and (y_1, y_2), respectively. If this is the case, the final address is computed by using ρ simply as an offset from (x, y), otherwise the x_t and/or y_t correctly addresses an out-of-block address.

$$\begin{aligned} T^x_y(\underline{t}, i, j) = {} & (\mathbf{if}\ x_1 \leqslant x_t \leqslant x_2\ \mathbf{then}\ x + i\ \mathbf{else}\ x_t, \mathbf{if}\ y_1 \leqslant y_t \leqslant y_2\ \mathbf{then}\ y + j\ \mathbf{else}\ y_t) \\ & \text{where } (x_1, x_2, w) = H^x(\underline{t}) \\ & \qquad\quad (y_1, y_2, h) \ = V_y(\underline{t}) \\ & \qquad\quad (x_t, y_t) \qquad = (x_1 + (x - x_1) \bmod w + i, y_1 + (y - y_1) \bmod h + j) \end{aligned}$$

The third step in computing references is to determine the range of cells for the originating cell (x, y) to account for expansion dependency. This *ignore range* is represented by four x and four y coordinates (since the block containing (x, y) must be omitted from the ignore range) and is computed through the function I by first determining the horizontal and vertical range for the referencing cell (x, y) and then removing the actual block that contains (x, y).

$$\begin{aligned} I^x_y(\underline{t}) = {} & ((x_0, x_1, x_2, x_3), (y_0, y_1, y_2, y_3)) \\ & \text{where } (x_0, x_3, w) = H^x(\underline{t}) \\ & \qquad\quad (y_0, y_3, h) \ = V_y(\underline{t}) \\ & \qquad\quad (x_1, x_2) \qquad = (x - x \bmod w - 1, x - x \bmod w + w) \\ & \qquad\quad (y_1, y_2) \qquad = (y - y \bmod h - 1, y - y \bmod h + h) \end{aligned}$$

Finally, with the functions T and I we can define the the function $\mathscr{R}$ to translate an offset into (a range of) absolute address(es).

$$\begin{aligned}\mathscr{R}_y^x(\underline{t}, i, j) = \{(k, l) \in X \times Y \mid k \bmod w = x' \bmod w \wedge l \bmod h = y' \bmod h\} \\ -\{(x, y)\} \cup \{(x', y')\} \\ \text{where } ((x_0, x_1, x_2, x_3), (y_0, y_1, y_2, y_3)) &= I_y^x(\underline{t}) \\ (x', y') &= T_y^x(\underline{t}, i, j) \\ (x_a, x_b, w) &= H^{x'}(\underline{t}) \\ (y_a, y_b, h) &= V_{y'}(\underline{t}) \\ X &= \{x_a, \ldots, x_b\} - (\{x_0, \ldots, x_1\} \cup \{x_2, \ldots, x_3\}) \\ Y &= \{y_a, \ldots, y_b\} - (\{y_0, \ldots, y_1\} \cup \{y_2, \ldots, y_3\})\end{aligned}$$

As an example, consider the template *SumCol* together with the instance $\underline{t}$ = Values^$0^{|3}$^$\Sigma(u)$. Assume we want to find the range of the u parameter in the bottom-most cell. In this case, we apply the function $\mathscr{R}_5^1(\underline{t}, 0, -1)$ since the u is in the fifth row of this instance, and it references an offset of one up in the vertical direction.

First, we compute $(x', y') = T_5^1(\underline{t}, 0, -1)$, which yields the horizontal and vertical information for the current area, giving us $x_1 = x_2 = 1$ and $y_1 = y_2 = 5$. Since the width and height are 1, x_t and y_t are simply given by $x_1 + i$ and $y_1 + j$ respectively, that is, $x_t = 1$ and $y_t = 4$. For x, we find that $x_1 \geqslant x_t \geqslant x_2$, so $x + i = 1$, is chosen as the x coordinate. For y, we find that $y_t = 4$ is outside of y_1 and y_2, which are both 5. So in this case, we select $y_t = 4$ as the y coordinate. The returned value is thus $(1, 4)$.

Next, the horizontal and vertical information of this target location is computed. We compute $H^1(\underline{t})$ and $V_4(\underline{t})$. This concludes that the target range has width 1, with $x_a = 1$ and $x_b = 1$. Vertically, the range is of height 3, with $y_a = 2$ and $y_b = 4$.

Next, we compute the ignore range, namely $I_5^1(\underline{t})$. This is also based on the horizontal and vertical information for $(1, 5)$. It computes x_0 and x_3 to be both 1, and y_0 and y_3 to be both 5. This is because the block is not repeating and is only 1×1 in size. (x_1, x_2) is found to be $(0, 2)$ and (y_1, y_2) is found to be $(4, 6)$. Thus the complete result is $((1, 0, 2, 1), (5, 4, 6, 5))$.

Now we can compute X and Y. X is the range from x_a to x_b excluding the ignore ranges x_0 to x_1, x_2 to x_3, and Y is the range from y_a to y_b excluding y_0 to y_1 and y_2 to y_3. Those four ranges are 1..0, 2..1, 5..4, and 6..5, respectively. All of these ranges are empty. Therefore, X and Y are simply the ranges $X = \{1\}$ and $Y = \{2, 3, 4\}$.

Finally, we compute all pairs from X and Y such that they align within the block (which is trivial in this case since the width and height are 1), excepting the origin (to avoid circular references) and including the original destination. This gives us the set $\{(1, 2), (1, 3), (1, 4)\}$.

As another example that illustrates how non-continuous ranges are constructed, consider the following template

$$(\mathsf{A} \mid \mathsf{B})\hat{}((0 \mid 0)\hat{}(0 \mid 0))^{\downarrow}\hat{}(\mathsf{Sum\ A} \mid \mathsf{Sum\ B})\hat{}(\Sigma(u^2) \mid \Sigma(u^2))$$

together with the following template instance.

$$\underline{t} = (\mathsf{A} \mid \mathsf{B})\hat{}((0 \mid 0)\hat{}(0 \mid 0))^{|2}\hat{}(\mathsf{Sum\ A} \mid \mathsf{Sum\ B})\hat{}(\Sigma(u^2) \mid \Sigma(u^2))$$

In this example we repeat vertically a square 2×2 block of numbers. At the bottom of this column, we have two sum fields, each one summing the lower halves of the 2×2 blocks in their column. Assume we want to find the range of the u^2 parameter in the leftmost sum. We apply $\mathscr{R}^1_7(\underline{t}, 0, -2)$.

To determine the range, the target location is determined by computing $T^1_7(\underline{t}, 0, -2)$. This tells us the horizontal and vertical information for the current area, giving us $x_1 = x_2 = 1$ and $y_1 = y_2 = 7$. Since the width and height are 1, x_t and y_t are simply $x_1 + i$ and $y_1 + j$, respectively. Thus, $x_t = 1$ and $y_t = 5$. For x, we find that $x_1 \geqslant x_t \geqslant x_2$, so $x + i = 1$, is chosen as the x coordinate. For y, we find that y_t, being 5, is outside of y_1 and y_2, which are both 7. So in this case, we select simply y_t, which is 5. Therefore, $T^1_7(\underline{t}, 0, -2)$ is $(1, 5)$.

Once the target location has been found, the horizontal and vertical information is calculated. We compute $H^1(\underline{t})$ and $V_5(\underline{t})$. This tells us that the target range has width 1, with $x_a = 1$ and $x_b = 1$. Vertically, the range is of height 2, with $y_a = 2$ and $y_b = 5$.

The horizontal and vertical information of the target location prepares us to construct the range of the reference, but before that, we must determine the ignore range, namely $I^1_7(\underline{t})$. In order to find this, we use the horizontal and vertical information for the origin location $(1, 7)$. The ignore range finds x_0 and x_3 to be both 1, and y_0 and y_3 to be both 7. This is because the block is not repeating and is only 1×1 in size. Next (x_1, x_2) is found to be $(0, 2)$ whereas (y_1, y_2) is found to be $(6, 8)$. Thus the complete result is $((1, 0, 2, 1), (7, 6, 8, 7))$.

At this point we are ready to determine the basic reference coordinate sets, X and Y. X is the range from $x_a = 1$ to $x_b = 1$ excluding the ignore ranges x_0 to x_1, x_2 to x_3, and Y is the range from $y_a = 2$ to $y_b = 5$ excluding y_0 to y_1 and y_2 to y_3. Those four ranges are 1..0, 2..1, 7..6, and 8..7, respectively, which are all empty. Therefore, X and Y are simply the ranges specified, namely $X = \{1\}$ and $Y = \{2, 3, 4, 5\}$.

In this case, the modulo in the final computation comes into play. Since the height is 2, only y values with the same modulo 2 as the target 5 will be accepted. In other words, only odd values of y. This excludes from the set the pairs $\{(1, 2), (1, 4)\}$, leaving us with the final range $\{(1, 3), (1, 5)\}$.

As a final example that demonstrates how references across aligned repeating blocks do *not* create ranges but single references, consider the template $0^{\downarrow} \mid \Sigma(\ell)^{\downarrow}$ and a corresponding template instance $0^{|3} \mid \Sigma(\ell)^{|3}$. In this example, we have two columns, each consisting of a vertical repeating block, concatenated horizontally. This is the interesting case of referencing from one repeating block to another which is aligned with the first. We will ask what the range of the first ℓ is, in cell $(2, 1)$. Therefore, we compute $\mathscr{R}^2_1(\underline{t}, -1, 0)$.

The target location, (x', y'), is determined first by applying $T^2_1(\underline{t}, -1, 0)$. In the process, this finds the horizontal and vertical information of the origin area, giving us $x_1 = x_2 = 2$ and $y_1 = 1$ and $y_2 = 3$, since there is a vertical repetition. Since the width and height are 1, x_t and y_t are simply $x_1 + i$ and $y_1 + j$ respectively, that is, $x_t = 1$ and $y_t = 1$. For x, we find that x_t is outside the range of x_1 to x_2, so we select x_t, which is 1, for our x coordinate. For y, we find that y_t is within the range of y_1 to y_2, so we accept $y + j$, which is 1. Therefore, the target location is determined to be $(1, 1)$.

The horizontal and vertical information of this target location reveals the possible extent of the reference. We use $H^1(\underline{t})$ and $V_1(\underline{t})$ to find it. These functions determine that the target range has width 1, with $x_a = 1$ and $x_b = 1$. Vertically, the range is of height 3, with $y_a = 1$ and $y_b = 3$.

We exclude from the possible extent of the reference anything indicated by the ignore range, $I_1^2(\underline{t})$. Using the horizontal and vertical information for $(2, 1)$ (the origin), it computes x_0 and x_3 to be both 2, and $y_0 = 1$ and $y_3 = 3$. This is because the block is vertically repeating. (x_1, x_2) is found to be $(1, 3)$ and (y_1, y_2) is found to be $(0, 2)$. The ignore range is $((2, 1, 3, 2), (3, 0, 2, 3))$.

Using these results, we can find X and Y. X is the range from x_a to x_b excluding the ignore ranges x_0 to x_1, x_2 to x_3, and Y is the range from y_a to y_b excluding y_0 to y_1 and y_2 to y_3. Notice that one of the ranges, 2..3, is not empty, but contains $\{2, 3\}$. These values must be excluded from the range of Y. This gives us the sets $X = \{1\}$ and $Y = \{1, 2, 3\} - \{2, 3\}$, with the final set for $Y = \{1\}$. Note that the values 2 and 3 have been excluded because the two columns align.

Since the block height and width is 1, the modulo does not come into the play, and the final reference set is simply $\{(1, 1)\}$.

3.2.3 Table generation

The translation function $\mathscr{G}$ to create tables from template instances defined in Figure 4 takes as input the complete template instance $(\underline{t})$ together with the position of the top-left corner of the part currently being translated (x, y), which is needed for the proper translation of references (third line). The last argument is the part of the template instance that is seen at the current location. The notation $\lceil S \rceil$ extracts the elements out of a set, that is, $\lceil \{x_1, \ldots, x_n\} \rceil = x_1, \ldots, x_n$. This technical adjustment is needed in the third line to inject the set of references computed by $\mathscr{R}$ as a single reference or sequence of references into a formula as required by the syntax of blocks.

$$\begin{aligned}
\mathscr{G}_y^x(\underline{t}, \phi) &= \phi \\
\mathscr{G}_y^x(\underline{t}, \phi(f_1, \ldots, f_n)) &= \phi(\mathscr{G}_y^x(\underline{t}, f_1), \ldots, \mathscr{G}_y^x(\underline{t}, f_n)) \\
\mathscr{G}_y^x(\underline{t}, (i, j)) &= \lceil \mathscr{R}_y^x(\underline{t}, i, j) \rceil \\
\mathscr{G}_y^x(\underline{t}, u \mid v) &= \mathscr{G}_y^x(\underline{t}, u) \mid \mathscr{G}_y^{x+\overleftrightarrow{u}}(\underline{t}, v) \\
\mathscr{G}_y^x(\underline{t}, u\hat{\ }v) &= \mathscr{G}_y^x(\underline{t}, u)\hat{\ }\mathscr{G}_{y+\updownarrow u}^x(\underline{t}, v) \\
\mathscr{G}_y^x(\underline{t}, c^{\underline{k}}) &= \mathscr{G}_y^x(\underline{t}, c) \mid \mathscr{G}_y^{x+\overleftrightarrow{c}}(\underline{t}, c) \mid \ldots \mid \mathscr{G}_y^{x+(k-1)\overleftrightarrow{c}}(\underline{t}, c) \\
\mathscr{G}_y^x(\underline{t}, b^{|k}) &= \mathscr{G}_y^x(\underline{t}, b)\hat{\ }\mathscr{G}_{y+\updownarrow b}^x(\underline{t}, b)\hat{\ }\ldots\hat{\ }\mathscr{G}_{y+(k-1)\updownarrow b}^x(\underline{t}, b)
\end{aligned}$$

Fig. 4. Table generation.

Applying the function $\mathscr{G}$ to $\mathscr{I}_1(t)$ yields the initial table, that is, a block that contains a copy of all the values and formulas from the template.

The function $\mathscr{G}$ generates a complete table consisting of only horizontal and vertically composed individual blocks from a template instance. A primary feature of this function is converting repeating groups into a sequence of individual blocks. It does so by breaking the table up piece by piece. In the case of a repeating group, either horizontal or vertical, $\mathscr{G}$ unrolls the group to actually be repeated that many times. It then uses the function $\mathscr{R}$ to determine the appropriate references to replace the relative offsets.

As an example, assume we want to construct the table for the instance $\underline{t} = \mathsf{Values}\hat{}0^{|3}\hat{}\Sigma(u)$. In this case, $\mathscr{G}$ first starts with $\mathscr{G}_1^1(\underline{t}, (\mathsf{Values}\hat{}0^{|3})\hat{}\Sigma(u))$. $\mathscr{G}$ breaks the template instance into the upper and lower segment, determines the height of the upper segment, and recursively calls $\mathscr{G}_1^1(\underline{t}, \mathsf{Values}\hat{}0^{|3})$ and $\mathscr{G}_5^1(\underline{t}, \Sigma(u))$. The latter parameter coordinate is generated by inspecting the height of the top piece, which consists of a single unit label and a three unit expansion block. The sum is calculated, $1 + (3 * 1)$ and the total value, 4, is added to the original offset of 1 to get a final offset of 5. The upper section is broken again, so that the repeating block is addressed with $\mathscr{G}_2^1(\underline{t}, 0^{|3})$, which leads to $\mathscr{G}_2^1(\underline{t}, 0)\hat{}\mathscr{G}_3^1(\underline{t}, 0)\hat{}\mathscr{G}_4^1(\underline{t}, 0)$. Each of these applications of $\mathscr{G}$ reduces to the argument value 0. The lower portion is handled by the case of function application, which turns $\mathscr{G}_5^1(\underline{t}, \Sigma(u))$ into $\Sigma(\mathscr{G}_5^1(\underline{t}, u))$, which results in $\Sigma((1,2),(1,3),(1,4))$, as shown through the example for illustrating the working of $\mathscr{R}$. Therefore, the generated table will be $\mathsf{Values}\hat{}0\hat{}0\hat{}0\hat{}\Sigma((1,2),(1,3),(1,4))$.

3.2.4 Update operations

Two kinds of update operations are allowed on generated tables: (1) changing values to other values of the same type[2] and (2) inserting and deleting rows and columns. The first kind of update is realized in the following way. Before a new value ϕ can be entered into a cell at address (x, y), it is ensured that the cell in the template that corresponds to (x, y) does not contain a formula and the type of the cell is the same as the type of ϕ. We write $chg_{(x,y)}^{\phi}(\underline{t}, b)$ for the update of the cell located at (x, y) in table b to the new value ϕ. The argument $\underline{t}$ gives the template instance that corresponds to b. Formally, $chg_{(x,y)}^{\phi}$ returns a pair $(\underline{t}, b')$ where $\underline{t}$ is the unchanged template instance and b' is the changed table. The effect of the row/column-insertion commands depends on the current position in the table. For example, the insert-column command will insert k new Excel columns if the current position is within a hex group that has the width k. The formulas and values to be inserted into the new cells are taken from the hex group of the template. For a position outside of a hex group the insert-column command has no effect. Similarly, the insert-row command works only when the current position is in a cell from a vex group, in which case k new rows will be inserted where k is the height of all the aligned vex groups covering the current vertical position. Again, formulas and values are copied from corresponding vex groups of the template.

In general, the insertion of columns and rows requires also the adjustment of absolute references in existing cells. We can accomplish the generation of absolute references in newly inserted formulas and the reference adjustments by employing the $\mathscr{G}$ function in the following way. First, we update the template instance by

[2] In fact, we allow arbitrary type-correct formulas that do not contain references.

increasing the exponent of a hex group (or a collection of vex groups). Then we can simply apply $\mathscr{G}$ to the new template instance and obtain correct formulas with correct absolute addresses for the whole table. Finally, we copy into this new table the values from the old table.

The functions for updating template instances are defined as follows. The functions C^x and R_y update a template instance on an insert-column or insert-row command, respectively. In these cases, both functions take a template instance and a current offset (x for column insert, y for row insert) to determine what, if anything, should be added. The functions find the location of the current position, and if it is within a vex or hex group, they increase the expansion of that group by one.

$$
\begin{aligned}
C^x(t \mid t') &= \begin{cases} C^x(t) \mid t' & \text{if } x \leqslant \overleftrightarrow{t} \\ t \mid C^{x-\overleftrightarrow{t}}(t') & \text{otherwise} \end{cases} \\
C^x(c^{\underline{k}}) &= \begin{cases} c^{\underline{k+1}} & \text{if } x \leqslant k\overleftrightarrow{c} \\ c^{\underline{k}} & \text{otherwise} \end{cases} \\
C^x(c) &= c \\
R_y(t \mid t') &= R_y(t) \mid R_y(t') \\
R_y(c^{\underline{k}}) &= (R_y(c))^{\underline{k}} \\
R_y(c\hat{}c') &= \begin{cases} R_y(c)\hat{}c' & \text{if } y \leqslant \updownarrow c \\ c\hat{}R_{y-\updownarrow c}(c') & \text{otherwise} \end{cases} \\
R_y(b^{|k}) &= \begin{cases} b^{|k+1} & \text{if } y \leqslant k\updownarrow b \\ b^{|k} & \text{otherwise} \end{cases} \\
R_y(b) &= b
\end{aligned}
$$

Consider again the template instance $\underline{t} = \mathsf{Values}\hat{}0^{|3}\hat{}\Sigma(u)$. Assume that an insert-row command is executed on row 2. In this case, we start with $R_2((\mathsf{Values}\hat{}0^{|3})\hat{}\Sigma(u))$. The height of the first block is 4, which is greater than the value for y (which is 2), so the function applies recursively to the first block, that is, we obtain $R_2(\mathsf{Values}\hat{}0^{|3})\hat{}\Sigma(u)$. The height of the first block is only 1, so the second case is executed, modifying the y parameter of R_y by the height of the top block. Therefore, the recursive call $R_1(0^{|3})$ results. We find that y is within the height of this block, so we increase its repetition, returning $0^{|4}$ as a subexpression, which leads to the final new template instance $\mathsf{Values}\hat{}0^{|4}\hat{}\Sigma(u)$.

Merging the actual values from the old table with the new table obtained by $\mathscr{G}$ is achieved by two functions that copy all values outside of the column (or row) range for the newly inserted column (row). These "ignore ranges" can be computed with the help of V and H because, after an insert column command, the x coordinate of the current position must be between $x_1 + kw$ and $x_1 + (k+1)w$ for some k where $H^x(\underline{t}) = (x_1, x_2, w)$. Similarly, the y coordinate must be between $y_1 + lh$ and $y_1 + (l+1)h$ for some l where $V_y(\underline{t}) = (y_1, y_2, h)$. The functions $\mathscr{H}$ and $\mathscr{V}$ can be defined as follows. They traverse the newly generated table and copy values from the old table within the old areas. $\mathscr{H}$ and $\mathscr{V}$ accept four parameters: an x and y coordinate, which both start at 1, along with a newly generated table and the old table (b) before the row or column insertion. The position of insertion is held as $\hat{x}$ and $\hat{y}$ where $\hat{x} = x_1 + kw$ and $\hat{y} = y_1 + lh$, so that we have $\hat{x} \leqslant x < \hat{x} + w$ or $\hat{y} \leqslant y < \hat{y} + h$ for the current position (x, y).

$$
\begin{aligned}
\mathscr{H}^x_y(b_1 \mid b_2, b) &= \mathscr{H}^x_y(b_1, b) \mid \mathscr{H}^{x+\overleftrightarrow{b_1}}_y(b_2, b) \\
\mathscr{H}^x_y(b_1{}^\wedge b_2, b) &= \mathscr{H}^x_y(b_1, b){}^\wedge \mathscr{H}^x_{y+\updownarrow b_1}(b_2, b) \\
\mathscr{H}^x_y(f, b) &= f \quad (\text{for } f \neq \phi) \\
\mathscr{H}^x_y(\phi, b) &= \begin{cases} \phi & \text{if } \hat{x} \leqslant x < \hat{x} + w \\ b[x, y] & \text{if } x < \hat{x} \\ b[x - w, y] & \text{if } x \geqslant \hat{x} + w \end{cases} \\
\mathscr{V}^x_y(b_1 \mid b_2, b) &= \mathscr{V}^x_y(b_1, b) \mid \mathscr{V}^{x+\overleftrightarrow{b_1}}_y(b_2, b) \\
\mathscr{V}^x_y(b_1{}^\wedge b_2, b) &= \mathscr{V}^x_y(b_1, b){}^\wedge \mathscr{V}^x_{y+\updownarrow b_1}(b_2, b) \\
\mathscr{V}^x_y(f, b) &= f \quad (\text{for } f \neq \phi) \\
\mathscr{V}^x_y(\phi, b) &= \begin{cases} \phi & \text{if } \hat{y} \leqslant y < \hat{y} + h \\ b[x, y] & \text{if } y < \hat{y} \\ b[x, y - h] & \text{if } y \geqslant \hat{y} + h \end{cases}
\end{aligned}
$$

Consider the template instance $\underline{t} = \mathsf{Values}{}^\wedge 0^{|3}{}^\wedge\Sigma(u)$ and a corresponding actual table $b = \mathsf{Values}{}^\wedge 1{}^\wedge 2{}^\wedge 3{}^\wedge\Sigma((1,2),(1,3),(1,4))$.

In this case, assume a row insert is made at position $(1, 3)$, that is, $\hat{x} = 1$ and $\hat{y} = 3$. The application of R_3 yields the new template instance $\mathsf{Values}{}^\wedge 0^{|4}{}^\wedge\Sigma(u)$ to which $\mathscr{G}$ is applied and produces the table

$$b' = \mathsf{Values}{}^\wedge 0{}^\wedge 0{}^\wedge 0{}^\wedge 0{}^\wedge\Sigma((1,2),(1,3),(1,4),(1,5))$$

as demonstrated previously. The function application $\mathscr{V}^1_1(b', b)$ unfolds into the two function calls $\mathscr{V}^1_1(\mathsf{Values}{}^\wedge 0{}^\wedge 0{}^\wedge 0{}^\wedge 0, b)$ and $\mathscr{V}^1_6(\Sigma((1,2),\ldots,(1,5)), b)$, which is unchanged since it is a formula (third rule of $\mathscr{V}$). The y parameter of 6 comes from the sum of the original 1 plus the calculated height of the first vertical block, which is 5. The block of the first function call is again broken vertically into $\mathscr{V}^1_1(\mathsf{Values}{}^\wedge 0{}^\wedge 0{}^\wedge 0, b)$ and $\mathscr{V}^1_5(0, b)$. The latter, whose parameter of 5 is computed the same way as above, is simply a value. Now the definition tries to find where in the original table the value should come from. In this case, the current y value of 5 is greater than the original $\hat{y}$ value of 3 plus the height of the inserted row, which is 1. Therefore, we access the location $(1, 5 - 1)$ (third case of the last rule for $\mathscr{V}$), which gives us the value 3 from the original table, which replaces the placeholder 0. The top block is broken again into $\mathscr{V}^1_1(\mathsf{Values}{}^\wedge 0{}^\wedge 0, b)$ and $\mathscr{V}^1_4(0, b)$. In this case, we find out that the value of y, being 4, is equal to the $\hat{y}$ plus the height of the row, which also totals to 4. In this case, we still apply the same rule as before, looking up $(1, 4 - 1)$ in the table, which gives us the value 2. The top block is broken again into $\mathscr{V}^1_1(\mathsf{Values}{}^\wedge 0, b)$ and $\mathscr{V}^1_3(0, b)$. In this case, the value of y, which is 3, is less than the value of $\hat{y}$ plus the height of the row. However, this is still equal to $\hat{y}$, so we take the placeholder value without looking one up in the original table b. This is where the "new" value appears.

The top block is broken again into $\mathscr{V}^1_1(\mathsf{Values}, b)$ and $\mathscr{V}^1_2(0, b)$. The latter case has a y which is less than $\hat{y}$, so simply the value of the $(1, 2)$ is looked up in the original table. The top value is analyzed likewise. Thus, the newly constructed table is

$$\mathsf{Values}\hat{\ }1\hat{\ }0\hat{\ }2\hat{\ }3\hat{\ }\Sigma((1,2),(1,3),(1,4),(1,5)).$$

Finally, the semantics of the insert-column and insert-row operation is defined as follows. In the given definitions, the $\underline{t}$ argument represents the current template instance, whereas the b argument represents the actual table. In addition to the new table, the functions also return the new template instance $\underline{t}'$.

$$ins^C_{(x,y)}(\underline{t}, b) = (\underline{t}', \mathscr{H}^1_1(\mathscr{G}^1_1(\underline{t}', \underline{t}'), b)) \quad \text{where } \underline{t}' = C^x(\underline{t})$$
$$ins^R_{(x,y)}(\underline{t}, b) = (\underline{t}', \mathscr{V}^1_1(\mathscr{G}^1_1(\underline{t}', \underline{t}'), b)) \quad \text{where } \underline{t}' = R_y(\underline{t})$$

Note that in the implemented Gencel system we do not keep a copy of the whole actual spreadsheet. Instead we send to Excel only cell definitions that need to be changed. The concept of template instances allows us to describe the update operations in the formal model as well as to implement a space efficient system.

Deleting rows and columns works in a similar way. First, we need two functions $\bar{R}$ and $\bar{C}$ for decreasing exponents, which are defined exactly as R and C, except for the exponent, which has to be $k-1$ instead of $k+1$ and is only decreased if $k > 1$. Moreover, we need two functions $\bar{\mathscr{H}}$ and $\bar{\mathscr{V}}$ that are defined like $\mathscr{H}$ and $\mathscr{V}$ except for the following cases.

$$\bar{\mathscr{H}}^x_y(\phi, b) = \begin{cases} b[x,y] & \text{if } x < \hat{x} \\ b[x+w, y] & \text{otherwise} \end{cases}$$

$$\bar{\mathscr{V}}^x_y(\phi, b) = \begin{cases} b[x,y] & \text{if } y < \hat{y} \\ b[x, y+h] & \text{otherwise} \end{cases}$$

For delete row and delete column we get the following definitions.

$$del^C_{(x,y)}(\underline{t}, b) = (\underline{t}', \bar{\mathscr{H}}^1_1(\mathscr{G}^1_1(\underline{t}', \underline{t}'), b)) \quad \text{where } \underline{t}' = \bar{C}^x(\underline{t})$$
$$del^R_{(x,y)}(\underline{t}, b) = (\underline{t}', \bar{\mathscr{V}}^1_1(\mathscr{G}^1_1(\underline{t}', \underline{t}'), b)) \quad \text{where } \underline{t}' = \bar{R}_y(\underline{t})$$

3.2.5 Table evaluation

The evaluation of a table essentially means to evaluate all cells by applying basic functions and looking up references. The evaluation of cells requires the whole table as an additional parameter to facilitate the evaluation of references, which are given by absolute addresses. The metavariable x used in the rules Sum and Prod ranges over numeric values.

As an example, consider the following table.

$$b = \mathsf{Values}\hat{\ }1\hat{\ }2\hat{\ }3\hat{\ }\Sigma((1,2),(1,3),(1,4))$$

The goal is to derive the table b' such that $b \twoheadrightarrow b'$, which can be achieved through the $\text{Tab}_{\twoheadrightarrow}$ rule.

$$\textsc{Val}_{\twoheadrightarrow}\ \frac{}{\phi \overset{b}{\twoheadrightarrow} \phi} \qquad \textsc{Sum}_{\twoheadrightarrow}\ \frac{f_k \overset{b}{\twoheadrightarrow} x_k \quad 1 \leqslant k \leqslant n}{\Sigma(f_1,\ldots,f_n) \overset{b}{\twoheadrightarrow} x_1 + \ldots + x_n}$$

$$\textsc{Prod}_{\twoheadrightarrow}\ \frac{f_k \overset{b}{\twoheadrightarrow} x_k \quad 1 \leqslant k \leqslant n}{\Pi(f_1,\ldots,f_n) \overset{b}{\twoheadrightarrow} x_1 * \ldots * x_n} \qquad \textsc{Ref}_{\twoheadrightarrow}\ \frac{b[\rho] = f \quad f \overset{b}{\twoheadrightarrow} \phi}{\rho \overset{b}{\twoheadrightarrow} \phi}$$

$$\textsc{Hor}_{\twoheadrightarrow}\ \frac{b_1 \overset{b}{\twoheadrightarrow} b_3 \quad b_2 \overset{b}{\twoheadrightarrow} b_4}{b_1 \mid b_2 \overset{b}{\twoheadrightarrow} b_3 \mid b_4} \qquad \textsc{Ver}_{\twoheadrightarrow}\ \frac{b_1 \overset{b}{\twoheadrightarrow} b_3 \quad b_2 \overset{b}{\twoheadrightarrow} b_4}{b_1 \hat{\ } b_2 \overset{b}{\twoheadrightarrow} b_3 \hat{\ } b_4} \qquad \textsc{Tab}_{\twoheadrightarrow}\ \frac{b \overset{b}{\twoheadrightarrow} b'}{b \twoheadrightarrow b'}$$

Fig. 5. Evaluation of tables (blocks).

Since b is a vertical composition of cells, we have to repeatedly apply the $\textsc{Ver}_{\twoheadrightarrow}$ rule, which causes the table to be broken into multiple individual chunks, namely the label Values, the numbers 1, 2 and 3, and the summation formula. These are all reduced individually, and then vertically concatenated. The label and the numbers are reduced immediately using the $\textsc{Val}_{\twoheadrightarrow}$ rule. This rule returns them unchanged.

To reduce the summation the premises of the $\textsc{Sum}_{\twoheadrightarrow}$ rule must be established. These preconditions require that all references from the $\textsc{Sum}_{\twoheadrightarrow}$ rule must already be reduced before the sum can be evaluated. In this case, the references refer to the three numbers, all of which have been reduced using the $\textsc{Val}_{\twoheadrightarrow}$ rule. The function application of the Σ function can then be reduced to $1+2+3$, which is 6. Therefore, the resulting table is Values^1^2^3^6.

4 Type system

In this section we define a type system for templates to guarantee a meaningful generation of tables and their update operations.

We distinguish between two sets of types. First, the types of formulas (φ) include base types (α), for example, *Num* and *String*, and (first-order) function types for functions with an arbitrary number of arguments. It is easy to add, for example, unary and binary operations and corresponding function types and additional function-application typing rules. Second, template types (τ) have the same structure as templates except that horizontal and vertical repetition are identified, see Figure 6.

$$\begin{array}{rcll} \varphi & ::= & \alpha \mid \alpha^+ \to \alpha & (\textit{formula types}) \\ \sigma, \tau & ::= & \alpha \mid \tau \mid \tau \mid \tau\hat{\ }\tau \mid \tau^+ & (\textit{templates types}) \end{array}$$

Fig. 6. Formula and template types.

The type system is defined through several judgments. First, we give typing rules for formulas. Since the type of a formula f depends, in general, on the types of formulas that are contained in cells referenced by f, we formalize the typing of formulas by a judgment $\sigma_y^x \rhd f : \varphi$ that expresses that f, found at position (x, y) in the template, has type φ in the context of the template type σ. The typing rules for formulas are shown in Figure 7. We have two rules for typing references that are used to distinguish between references to single cells and ranges. We can reuse the functions defined in Section 3.2.2 to determine the nature of a reference ρ. First of all, we determine whether or not a referenced cell $b = (x', y')$ is in a hex or vex group, because only then it can mean a range. To this end, we can check the range computed by H or V for a template instance in which each repeating group has been expanded at least twice (which can be obtained by $\mathscr{I}_2(t)$): If the spanned cell range is larger than the width of the block, the cell is located in a repeating group. Second, the referenced cell denotes a range if and only if its repeating group is independent of the referencing cell, which is the case only if its range is different. Therefore, we can define the “is-range” predicate Θ as follows.

$$\Theta_y^x(t,(i,j)) = (x_2' - x_1' > w \wedge (x_1', x_2') \neq (x_1, x_2)) \vee (y_2' - y_1' > h \wedge (y_1', y_2') \neq (y_1, y_2))$$

$$\begin{array}{lll} \text{where } \underline{t} & = & \mathscr{I}_2(t) \\ (x', y') & = & T_y^x(\underline{t}, i, j) \\ (x_1', x_2', w) & = & H^{x'}(\underline{t}) \\ (y_1', y_2', h) & = & V_{y'}(\underline{t}) \\ (x_1, x_2, \_) & = & H^x(\underline{t}) \\ (y_1, y_2, \_) & = & V_y(\underline{t}) \end{array}$$

In rule App we use the notation $\alpha^{[+]}$ to represent α or α^+, which allows single references as well as range references to be used as function arguments. However, range references are otherwise prohibited in cells. This restriction is expressed effectively through the Fml rule in Figure 9, which requires α and prohibits α^+ for f.

$$\textsc{Val}\ \frac{\phi \text{ has type } \varphi}{\sigma_y^x \rhd \phi : \varphi} \qquad \textsc{App}\ \frac{\sigma_y^x \rhd \phi : \alpha^+ \to \alpha' \qquad \sigma_y^x \rhd f_i : \alpha^{[+]}}{\sigma_y^x \rhd \phi(f_1, \ldots, f_n) : \alpha'}$$

$$\textsc{Ref}\ \frac{\sigma[(x,y)+\rho] = \tau \qquad \neg\Theta_y^x(\sigma,\rho)}{\sigma_y^x \rhd \rho : \tau} \qquad \textsc{Ref}^+\ \frac{\sigma[(x,y)+\rho] = \tau \qquad \Theta_y^x(\sigma,\rho)}{\sigma_y^x \rhd \rho : \tau^+}$$

Fig. 7. Formula typing rules.

We do not allow the arbitrary alignment of blocks and columns. Some constraints are already expressed by the abstract syntax. In addition, we allow the vertical composition only for blocks of equal width, see the rules Ver and Col in Figure 9. Finally, we restrict the horizontal composition to columns that have the same vertical *pattern*. This constraint is expressed through the alignment predicate $t \wr t$, which is formalized in Figure 8 and which is used in rule Template in Figure 9.

$$\frac{}{t \wr t} \qquad \frac{t \wr t'}{t' \wr t} \qquad \frac{t_1 \wr t_2 \qquad t_2 \wr t_3}{t_1 \wr t_3} \qquad \frac{t_1 \wr t \qquad t_2 \wr t}{t_1 \mid t_2 \wr t} \qquad \frac{c \wr t}{c^{\to} \wr t} \qquad \frac{c_1 \wr c_3 \qquad c_2 \wr c_4}{c_1\hat{\ }c_2 \wr c_3\hat{\ }c_4}$$

$$\frac{c_1 \wr c_4 \qquad c_2 \wr c_5 \qquad c_3 \wr c_6}{c_1\hat{\ }(c_2\hat{\ }c_3) \wr (c_4\hat{\ }c_5)\hat{\ }c_6} \qquad \frac{b_1 \wr b_2}{b_1^{\downarrow} \wr b_2^{\downarrow}} \qquad \frac{\updownarrow b_1 = \updownarrow b_2}{b_1 \wr b_2}$$

Fig. 8. Column alignment.

The first three rules in Figure 8 define that vertical alignment is an equivalence relation. The next rule expresses that vertical alignment holds for horizontally composed templates if it holds for both templates individually. The fifth rule states that vertical alignment is invariant under horizontal repetition. The sixth rule defines vertical alignment as a congruence relation with respect to vertical composition while the seventh rule expresses that vertical composition is associative with respect to the equivalence defined by vertical alignment. The eighth rule defines that vertical alignment hold for vex groups if it holds for the argument blocks, and the last rule establishes the base case that says that blocks align vertically if the have the same height.

The typing rules for templates shown in Figure 9 define judgments of the form $\sigma_y^x \vdash t : \tau$. We overload the judgment notation for blocks, columns, and tables.

$$\textsc{Fml}\ \frac{\sigma_y^x \rhd f : \alpha}{\sigma_y^x \vdash f : \alpha} \qquad \textsc{Hor}\ \frac{\sigma_y^x \vdash b : \tau \quad \sigma_y^{x+\overleftrightarrow{b}} \vdash b' : \tau' \quad \updownarrow b = \updownarrow b'}{\sigma_y^x \vdash b \mid b' : \tau \mid \tau'}$$

$$\textsc{Ver}\ \frac{\sigma_y^x \vdash b_1 : \tau \quad \sigma_{y+\updownarrow b_1}^x \vdash b_2 : \tau' \quad \overleftrightarrow{b_1} = \overleftrightarrow{b_2}}{\sigma_y^x \vdash b_1 \hat{} b_2 : \tau \hat{} \tau'}$$

$$\textsc{Block}^+\ \frac{\sigma_y^x \vdash b : \tau}{\sigma_y^x \vdash b^{\downarrow} : \tau^{\downarrow}} \qquad \textsc{Col}\ \frac{\sigma_y^x \vdash c_1 : \tau \quad \sigma_{y+\updownarrow c_1}^x \vdash c_2 : \tau' \quad \overleftrightarrow{c_1} = \overleftrightarrow{c_2}}{\sigma_y^x \vdash c_1 \hat{} c_2 : \tau \hat{} \tau'}$$

$$\textsc{Col}^+\ \frac{\sigma_y^x \vdash c : \tau}{\sigma_y^x \vdash c^{\rightarrow} : \tau^{\rightarrow}} \qquad \textsc{Template}\ \frac{\sigma_y^x \vdash t : \tau \quad \sigma_y^{x+\overleftrightarrow{\tau}} \vdash t' : \tau' \quad t \wr t'}{\sigma_y^x \vdash t \mid t' : \tau \mid \tau'}$$

Fig. 9. Table typing rules.

To illustrate the typing rules, we give a couple of examples. Using rule Val (from Figure 7) and Block$^+$ (from Figure 9), we can derive that $0^{\downarrow}$ has type $Num^{\downarrow}$. Since Values has type *String* and is also of width 1, rule Ver can be employed to show that Values^$0^{\downarrow}$ has type $String\hat{}Num^{\downarrow}$. To type the reference in the formula $\Sigma(u)$ we need a template-type context. With a context $\sigma = String\hat{}Num^{\downarrow}\hat{}Num$ we can first derive by rule Ref$^+$ $\sigma_3^1 \rhd u : Num^{\downarrow}$ (the row number 3 results from the adjustment in the second premise of the Ver rule). Since according to rule Val, Σ has the type $Num^+ \rightarrow Num$ in any template-type context σ_y^x, we can apply the App rule to obtain the type *Num* for the summation cell, which finally yields the type σ for the whole summation column.

For the type of the summation template shown at the end of section 3.1 we first determine the type for the hex group, which according to rule Col$^+$ is $\sigma^{\rightarrow}$ (where σ is the type of the summation column). For typing the formula $\Sigma(\ell)$ in the total column, we again need a context, which we select as $\tau = \sigma^{\rightarrow} \mid \sigma$.

The type system allows the typing of cyclic references by assuming a fixed, arbitrary type for all cells on the cycle in σ_y^x. Cycles represent nonterminating computations and correspond to nonterminating function definitions whose value is undefined. In contrast to a Turing-complete functional language, we can easily detect nonterminating computations by identifying cycles in templates. Therefore, we consider a template to be type correct only if it does not contain any cycles. Although we could encode the cycle detection (or better, prevention) into the type system, it seems to be easier to add an explicit definition. We can determine all references that are contained in a formula by the following function.

$$\begin{aligned}
\Re(\phi) &= \varnothing \\
\Re(\rho) &= \{\rho\} \\
\Re(\phi(f_1, \dots, f_n)) &= \cup_{1 \leqslant i \leqslant n} \Re(f_i)
\end{aligned}$$

A template t contains a *cycle* $[(x_1, y_1), \dots, (x_n, y_n)]$ iff

$$\forall 1 \leqslant i < n.(x_{i+1}, y_{i+1}) \in \Re(t[x_i, y_i]) \wedge (x_n, y_n) \in \Re(t[x_1, y_1])$$

Next we define the notion of well typing for templates (and tables).

Definition 4.1
t is well typed with template type τ if $\tau_1^1 \vdash t : \tau$ and t does not contain a cycle.

5 Evolution safety

The main result for the presented table calculus is that a type-correct template allows only the generation of tables that can be always safely evaluated and never result in a computational error, such as a type error or reference error. To express this result formally we define the set of tables $\mathscr{T}(t)$, that is, the set of (template instance, table) pairs, that can be obtained from a template t through update operations as follows. $\mathscr{T}(t)$ is the smallest set satisfying:

$$\begin{aligned}
&(1) \quad (\mathscr{I}_1(t), \mathscr{G}_1^1(\mathscr{I}_1(t), \mathscr{I}_1(t))) \in \mathscr{T}(t) \\
&(2) \quad (\underline{t}, b) \in \mathscr{T}(t) \implies u_{(x,y)}(\underline{t}, b) \in \mathscr{T}(t) \\
&\qquad \text{for } 1 \leqslant x \leqslant \overleftrightarrow{b}, 1 \leqslant y \leqslant \updownarrow b \text{ and} \\
&\qquad u \in \{ins^R, ins^C, del^R, del^C, chg^\phi\}
\end{aligned}$$

We use the judgment $\Downarrow b$, which is defined in Figure 10, to express that the table b is fully evaluated, that is, b contains only values and does not contain any unevaluated formulas or unresolved references.

$$\frac{}{\Downarrow\phi} \qquad \frac{\Downarrow b \quad \Downarrow b'}{\Downarrow(b\,\hat{}\,b')} \qquad \frac{\Downarrow b \quad \Downarrow b'}{\Downarrow(b \mid b')}$$

Fig. 10. Table normal form.

The safety result can now be expressed as follows.

Theorem 1
If t is well typed and $(\underline{t}, b) \in \mathscr{T}(t)$, then $\exists b'.b \twoheadrightarrow b' \wedge \Downarrow b'$.

Proof
(Sketch) The proof is by induction over the construction history for elements of $\mathscr{T}(t)$, that is, first we show that the theorem is true for $b_t = \mathscr{G}_1^1(\mathscr{I}_1(t), \mathscr{I}_1(t))$. Then we show that each application of an update operation preserves the property. The theorem follows then by induction over the number of applied updates.

(1) It is obvious from the definition of $\mathscr{G}$ that b_t differs from t only in the following way: For any subexpression u^+ in t, the corresponding subexpression in b_t is just u, because the exponent is translated by $\mathscr{I}$ into 1, which is simply ignored by $\mathscr{G}$. The evaluation of b_t works by recursively descending to formulas (rules $\textsc{Hor}_{\twoheadrightarrow}$ and $\textsc{Ver}_{\twoheadrightarrow}$ in Figure 5). Values can always be evaluated to themselves (rule $\textsc{Val}_{\twoheadrightarrow}$). The rules for function application (rules $\textsc{Sum}_{\twoheadrightarrow}$ and $\textsc{Prod}_{\twoheadrightarrow}$) require that all arguments can be evaluated to values of the appropriate type. This precondition is ensured by the typing rule $\textsc{App}$ that is shown in Figure 7. The evaluation of a reference requires that the referenced cell exists and that the contained formula can be evaluated (rule $\textsc{Ref}_{\twoheadrightarrow}$ in Figure 5). The existence of references is guaranteed by the typing rules $\textsc{Ref}$ and $\textsc{Ref}^+$ in Figure 7. The fact that the referenced formula can be evaluated follows by induction because the well typing of t implies the absence of cycles.

(2) Next we consider an arbitrary element $(\underline{t}, b) \in \mathscr{T}(t)$. Let $(\underline{t}', b') = ins^C_{(x,y)}(\underline{t}, b)$. First, C^x increases the index of the hex group that covers the x coordinate by one. Then $\mathscr{G}$ regenerates the formulas for the whole new table. The adjustment of the x coordinate by $\overleftrightarrow{c}$ ensures that the generator keeps track of the correct position for the generation of all instances of the column, in particular, the newly inserted one and the ones that are moved to the right. This fact guarantees (through the definition of $\mathscr{R}^x_y$) that all references for newly generated cells and for moved cells will be translated into absolute addresses that refer to cells of the same type as the relative references in the template. Since the definition of $\mathscr{H}$ takes into account the position and width of the newly inserted column, the process of copying the values from the old table into the generated table does not change any reference. Therefore, the same line of reasoning as under (1) applies to the evaluation of references and arguments of operations, only on a larger set of cells.

An initial spreadsheet is known to be free of circular references. For each vex or hex group, it may reference some aligned and some unaligned blocks. If a row or column is added, a reference to an aligned block will be of the same form, thus not introducing a new circular reference. A reference to an unaligned block will remain constant. That unaligned block cannot reference the newly inserted cell, because it may only reference the entire vex/hex group or none of it. If it did reference the group, then there would have already been a circular reference in the initial sheet. If it does not, then no new circular reference will be introduced by the inserted cell referencing it.

Therefore, the theorem is true for the ins^C operation. Similar considerations apply to the other structure-changing operations. Since the chg^ϕ operation changes a value to another value of the same type, the theorem is true also for this operation. □

In analogy to Milner's slogan that "well-typed programs cannot go wrong", the above result can be paraphrased as "well-typed templates cannot evolve wrong".

6 Implementation of the Gencel system

The components of the Gencel system are shown in Figure 11. The generator and type checker are implemented in Haskell (Peyton Jones, 2003). These components

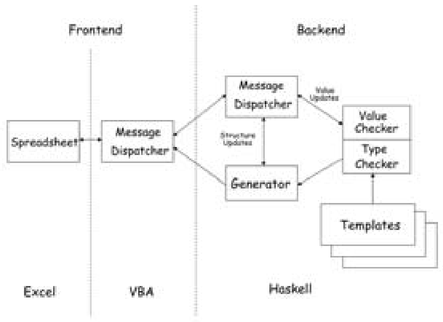
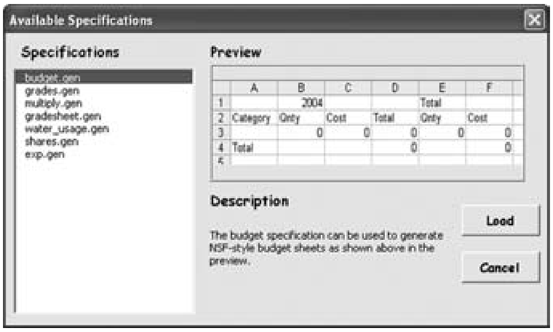


Fig. 11. Gencel system architecture.



budget.gen
grades.gen
multiply.gen
gradesheet.gen
water_usage.gen
shares.gen
exp.gen

	A	B	C	D	E	F
1		2004			Total	
2	Category	Qnty	Cost	Total	Qnty	Cost
3		0	0	0	0	0
4	Total			0		0
5						

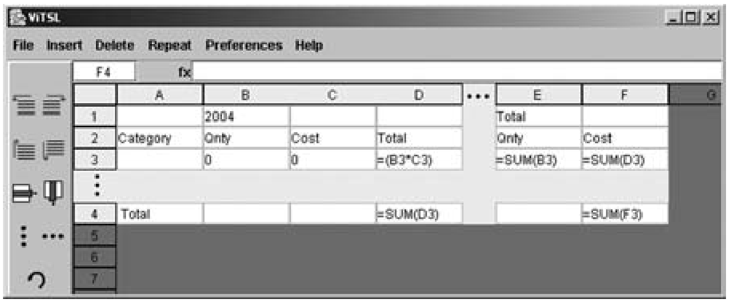
The budget specification can be used to generate NSF-style budget sheets as shown above in the preview.

Fig. 12. Interface for loading Gencel specifications.

are connected through a VBA module to Excel, which serves mainly as the user interface. We have already successfully employed a similar Haskell-backend strategy in the recent implementation of a header and unit inference system (Abraham & Erwig, 2004) and a debugger (Abraham & Erwig, 2005) for Excel. The information from the Excel sheet being manipulated by the end user is captured by a VBA program and sent to the backend server. The VBA system is shipped as an Excel add-in. The Haskell modules are compiled with GHC (GHC, 2004) to a Windows executable that runs as the backend server.

The Gencel toolbar has four buttons for row and column insertions, two buttons for row and columns deletions, and one button to bring up the interface for loading the specifications. This interface (shown in Figure 12) shows the user a listing of the available templates. When the user clicks on any file name from the list on the left she is shown a preview and a description of the specification, which can be added manually as a comment to templates.

Depending on which button has been clicked, the VBA program sends the corresponding message, with information about the current cell selection, to the backend server. The server performs the update to yield the new template instance. It then generates the messages for the updates to be performed to the Excel spreadsheet and sends them to the VBA program (these messages simply *paint* the new template instance in the Excel spreadsheet). Through Excel events, the VBA program also keeps track of value updates to the Excel spreadsheet.



	A	B	C	D	...	E	F	G
1		2004				Total		
2	Category	Qnty	Cost	Total		Qnty	Cost	
3		0	0	=(B3*C3)		=SUM(B3)	=SUM(D3)	
⋮								
4	Total			=SUM(D3)			=SUM(F3)	
5								
6								
7								

Fig. 13. ViTSL interface.

The backend server contains a "Message Dispatcher" that receives messages from the VBA program. In case of structural updates like row/column insert/delete operations, the dispatcher interacts with the "Generator" module to come up with the new template instance and the messages to reflect the update in the Excel frontend. In case the user changes a value in the Excel spreadsheet, the dispatcher forwards the message to the "Value Checker" module that checks the updated value against the specification to ensure that the new value is type correct. The other components of the backend server include the "Type Checker" module that checks the template loaded into the system by the user to ensure that it is type correct. The template, after type checking, is translated into the initial template instance and table by the generator.

The frontend VBA program keeps a copy of the ViTSL representation of the most recent template instance. Each time the user issues the save-file command in Excel, Excel saves the workbook and the corresponding ViTSL template instance. Whenever the Gencel add-in has been enabled in Excel, every time the user loads a workbook in Excel, the events trigger the backend server to load the corresponding template instance. This allows us to keep both files synchronized.

Moreover, we have implemented an interface that allows users to edit ViTSL specifications. The interface is shown in Figure 13 with the template for the budget sheet.

ViTSL is targeted at domain experts who are familiar with Excel. Therefore, to maintain the closeness of mapping, we have adopted a structure and behavior similar to Excel. Along the lines of the Gencel system, the ViTSL interface also allows insertion of rows (or columns) above or below (left or right) of the current cell. We have additional buttons for the creation of vertically or horizontally repeating groups. The interface differs from the visual notation used in section 2 in how it represents repeating groups. In the interface, the light gray shading marks the expansion areas of repeating groups, and the three vertical and horizontal dots (for vertically/horizontally repeating groups) appear only on the borders. Moreover, the absence of separators between the cells in the headers expresses that those rows or columns are part of the same repeating group. For example, in Figure 13, columns B, C, and D belong to the same repeating group.

The design of the ViTSL editor and an evaluation of the design using the Cognitive Dimension framework (Green & Petre, 1996) is described in more detail in Abraham *et al.* (2005).

7 Related work

The pervasiveness of errors in spreadsheets has motivated some research into spreadsheet design (Isakowitz *et al.*, 1995; Yoder & Cohn, 1994; Ronen *et al.*, 1989), testing (Rothermel *et al.*, 2001), consistency checking (Erwig & Burnett, 2002; Burnett & Erwig, 2002; Burnett *et al.*, 2003; Ahmad *et al.*, 2003; Antoniu *et al.*, 2004; Abraham & Erwig, 2004), fault localization (Ruthruff *et al.*, 2003; Prabhakarao *et al.*, 2003), and debugging (Abraham & Erwig, 2005).

However, little research has been performed on creating new, safer spreadsheet systems. The spreadsheet language Forms/3 (Burnett *et al.*, 2001) extends the spreadsheet paradigm by a number of features found in other programming languages. The language contains many experimental features, such as a time dimension, generalizations, gestures, and a model of sequence I/O. The application is itself not so much an end-user application, but a platform to study potential end-user applications. Since Forms/3 allows the free positioning and resizing of cells, it deviates from the traditional interaction model of spreadsheets, which makes it difficult to implement it as an extension of, for example, Excel as we did. Our approach is also strongly concerned with typing and safety, whereas this was not the primary concern in the *design* of the Forms/3 system.

On the other hand, the Forms/3 group has explored different approaches for auditing spreadsheets. Rothermel et al. have come up with the “What You See Is What You Test” methodology for testing spreadsheets (Rothermel *et al.*, 2001). This methodology uses data flow adequacy criteria and coverage monitoring to give users incremental feedback (using cell coloring and a “testedness” progress bar) on the percentage of cells that have been tested. Fisher et al. have developed a system that automatically generates test cases for end users to test their spreadsheets (Fisher II *et al.*, 2002). Fault localization techniques to help end users debug spreadsheets have also been incorporated into the Forms/3 language (Ruthruff *et al.*, 2003; Prabhakarao *et al.*, 2003).

The formulae, formats, relations (FFR) model presented in Sajaniemi (2000) abstracts the structure of spreadsheets. It also helps to analyze visualization mechanisms for spreadsheets. In this approach, errors in spreadsheet formulas show up as anomalies in the visualizations. A similar approach of identifying recurring structure (regions) in spreadsheets and then presenting anomalies as potential problem areas to the user is followed by the system described in Mittermeir & Clermont (2002). Both systems are designed to provide insights into the structure of existing spreadsheets; they are not intended to prevent errors in spreadsheets.

The Haxcel system of Lisper & Malmström (2002) is an attempt to bring the advantages of spreadsheets together with the type-safe, functional language Haskell. They define a system where Haskell programs are defined in a graphical list, and specialized arrays of data are displayed as tables. Any changes to the Haskell program will be immediately visible in the results of computation. This creates a tight “definition-eval-display loop” as spreadsheets have. Our approach is to begin at spreadsheets and move them toward a type-safe, structured system. Haxcel uses the opposite approach: to begin with a type-safe, structured language (Haskell) and make it more like a spreadsheet. They do this by separating the Haskell code from the data it is to process and placing the data in a table. In both cases, the goal of providing users with a clean and safe environment is present. However, very few users are familiar with Haskell compared to the number familiar with spreadsheets and this discrepancy is a potential barrier to the adoption of the Haxcel system. On the other hand, our system allows users to use functions and formulas that they are comfortable with in a familiar environment.

The FunSheet system (de Hoon *et al*., 1995) is another work which combines spreadsheets and general functional programming. The FunSheet system uses a regular row and column spreadsheet, but replaces the traditional formulas with more expressive functional syntax. The cells are lazily evaluated and may contain lists and other composite values. Cells are referenced by the use of column functions, that is, each column (A, B, etc.) is a function which takes an integer parameter and returns a cell value. Cell formulas can include functions like map and fold. The system also allows partial reductions of cell formulas and unbound variables. FunSheet is implemented in Clean and is used as an experiment in functional graphics and I/O as well as spreadsheets. FunSheet and our approach are similar on the surface: both start with a row and column spreadsheet model and move toward a safer system. FunSheet retains the unstructured model of spreadsheets and instead allows highly expressive functional programming in the cells. Our system retains many of the familiar abilities of spreadsheet formulas, but enforces a structured layer on the sheet itself. This is quite different from the FunSheet approach. One reason for this difference is a difference in goals: FunSheet seeks to explore and extend the computational capabilities of spreadsheets while our system seeks to increase safety and usability of spreadsheets.

The approach of Peyton Jones and others (Peyton Jones *et al*., 2003) to extend Excel by user-defined functions has a strong basis in end-user usability. The authors' primary goal is to design a method for end users to easily create new functions in their spreadsheets. They use a familiar spreadsheet model and attempt to minimize the impact (learning curve) on users while maximizing the user's productivity. The authors note that traditional spreadsheets often compute very complex models, but do not include any user-defined functions, which makes the spreadsheets needlessly complicated and difficult to maintain. The authors introduce a system that allows users to define functions within the existing spreadsheet model. The user creates a function sheet, which appears the same as any ordinary sheet, and defines the function from input cell(s) to an output cell. They may then use the function like any other built-in function. The authors have implemented a minimal prototype using VBA macros in Excel. They have also devised several mock-ups to demonstrate the concept of user-defined functions in Excel. The approach of the authors differs strongly from our approach. First, the authors focus heavily on the cognitive dimensions approach (Green & Petre, 1996). The authors also use as much of the existing paradigm as possible – they purposefully do not attempt to add a new representation for functions or new structure to the spreadsheets. The authors' approach does not increase the safety of a spreadsheet, except perhaps in cases where functions decrease the complexity of a sheet and allow the user to notice errors. However, the approaches are not entirely different. Both the authors' and our approach start with a familiar spreadsheet model. Both seek to improve end-user experience specifically. Both are more concerned with usability than adding new features to the spreadsheet system.

8 Conclusion and future work

We have designed a specification language for describing spreadsheet tables and their possible evolutions. The language is based on a table evolution calculus that defines semantics of generating tables from templates and the evaluation of tables. We have developed a type system for this calculus that ensures that any table obtained from a template can be safely evaluated without causing any errors.

We have implemented a prototype, called Gencel, as an extension to Excel that allows users to work safely with tables based on templates. The current system works well for many examples and can even conveniently deal with some computations that are difficult to perform in Excel.

In particular, Gencel exterminates the following kinds of errors from spreadsheets.

- *Range errors* (for example, omitted cells in aggregations)
- *Reference errors*
- *Type errors*

The impact of these errors has been extensively documented. For example, an omission error has caused a Florida construction company to underbid a project by a quarter of a million dollars (Ditlea, 1987; Hayen & Peters, 1989; Gilman & Bulkeley, 1986). An example of a type error is the illegal interpretation of a date as a numeric value, which caused an operating fund of the Colorado Student Loan Program to be understated by $36,131 (U.S. Department of Education, 2003). Finally, a reference error caused a hospital's records to overstate its Medicaid/Medicare crossover log by $38,240 (U.S. Department of Health and Human Services, 2003). The use of Gencel would have prevented all these errors.

The presented formalism and implementation indicate a new direction for functional programming research that is targeted at end users. Due to the widespread use of spreadsheets, the impact of work in this area can be enormous.

We have identified several topics for future research, both relating to the formal system and the tool implementation. We first discuss extensions to the formal system.

Each value ϕ in a cell of an expandable group potentially denotes a sequence of values, because the insert row/column operations can duplicate the cell. In the current model, this sequence of values is fixed to be $[\phi, \phi, \phi, \phi, \dots]$. A simple and useful generalization is to allow the specification of *value generators* in cells of expandable groups.

Currently, references all relate from one cell to another within the same table. If spreadsheets were allowed to contain multiple tables, it should be possible for references to refer to other tables in the same spreadsheet. There are several possible approaches to construct multiple-table spreadsheets. Perhaps the simplest approach would be to name tables and use names plus relative references.

Another limitation of references in expandable groups is that they either refer to another cell in the current repeating group or a fixed cell or range outside of the current group. No mechanism exists to allow a recursive reference to a different expansion of the same group. To solve this problem, the system needs a new kind of relative references, which reference a different generated block within the same expansion area. Such an extension affects syntax and semantics of the table calculus. Moreover, the rules of the type system become more complex since the references may be constructed to float over a wide variety of cells as expansion occurs.

Future work also extends into improvements of Gencel and associated tools. To support a wide-spread use of the Gencel system and to enable a smooth transition, we have to offer tools that can load existing Excel spreadsheets into the Gencel system. Since those spreadsheets are only given in Excel format, we have to distill a template of which they can be an instance. We call this process of identifying templates from plain spreadsheets *template parsing*. Template parsing is indispensable for the work with legacy spreadsheets.

It might not be possible to completely automate this process because of ambiguities. Techniques employed in probabilistic grammars (Charniak, 1996) might be useful to generate a ranked list of possible parsed templates. Alternatively, we could adapt spatial analysis techniques to identify semantic structures in spreadsheets that we developed to support the automatic header and unit inference (Abraham & Erwig, 2004).

Another feature that seems to be valuable in practice is the ability to encode formatting information with a spreadsheet template. This does not pose any theoretical difficulties.

Acknowledgments

The authors would like to thank the anonymous reviewers for their help in improving the presentation of this paper.

References

Abraham, R. and Erwig, M. (2004) Header and Unit Inference for Spreadsheets Through Spatial Analyses. *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pp. 165–172.

Abraham, R. and Erwig, M. (2005) Goal-Directed Debugging of Spreadsheets. *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*. pp. 189–196.

Abraham, R., Erwig, M., Kollmansberger, S. and Seifert, E. (2005) Visual Specifications of Correct Spreadsheets. *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*. pp. 37–44.

Ahmad, Y., Antoniu, T., Goldwater, S. and Krishnamurthi, S. (2003) A Type System for Statically Detecting Spreadsheet Errors. *18th IEEE Int. Conf. on Automated Software Engineering*, pp. 174–183.

Antoniu, T., Steckler, P. A., Krishnamurthi, S., Neuwirth, E. and Felleisen, M. (2004) Validating the Unit Correctness of Spreadsheet Programs. *26th IEEE Int. Conf. on Software Engineering*, pp. 439–448.

Brown, P. S. and Gould, J. D. (1987) An Experimental Study of People Creating Spreadsheets. *ACM Trans. Office Infor. Syst.* **5**(3), 258–272.

Burnett, M. M. and Erwig, M. (2002) Visually Customizing Inference Rules About Apples and Oranges. *2nd IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pp. 140–148.

Burnett, M. M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J. and Yang, S. (2001) Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *J. Funct. Program.*, **11**(2), 155–206.

Burnett, M. M., Cook, C., Summet, J., Rothermel, G. and Wallace, C. (2003) End-User Software Engineering with Assertions. *25th IEEE Int. Conf. on Software Engineering*, pp. 93–103.

Charniak, E. (1996) *Statistical Language Learning*. MIT Press.

de Hoon, W. A. C. A. J., Rutten, L. M. W. J. and van Eekelen, M. C. J. D. (1995) Implementing a Functional Spreadsheet in Clean. *J. Funct. Program.* **5**(3), 383–414.

Ditlea, S. (1987) Spreadsheets Can be Hazardous to Your Health. *Personal Comput.* **11**(1), 60–69.

Erwig, M. and Burnett, M. M. (2002) Adding Apples and Oranges. *4th Int. Symp. on Practical Aspects of Declarative Languages*, pp. 173–191. LNCS 2257.

Erwig, M., Abraham, R., Cooperstein, I. and Kollmansberger, S. (2005) Automatic Generation and Maintenance of Correct Spreadsheets. *27th IEEE Int. Conf. on Software Engineering*, pp. 136–145.

Fisher II, M., Cao, M., Rothermel, G., Cook, C. and Burnett, M. M. (2002) Automated Test Case Generation for Spreadsheets. *24th IEEE Int. Conf. on Software Engineering*, pp. 141–151.

GHC. (2004) *The Glasgow Haskell Compiler*. `http://haskell.org/ghc`.

Gilman, H. and Bulkeley, W. (1986) Can Software Firms be Held Responsible When a Program Makes a Costly Error? *Wall Street J.* **CCVII**(24), 17.

Green, T. R. G. and Petre, M. (1996) Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *J. Visual Lang. & Comput.* **7**(2), 131–174.

Hayen, R. L. and Peters, R. M. (1989) How to Ensure Spreadsheet Integrity. *Management Accounting*, **60**(9), 30–33.

Hendry, D. G. and Green, T. R. G. (1994) Creating, Comprehending and Explaining Spreadsheets: A Cognitive Interpretation of What Discretionary Users Think of the Spreadsheet Model. *Int. J. Human-Computer Stud.* **40**, 1033–1065.

Isakowitz, T., Schocken, S. and Lucas, Jr., H. C. (1995) Toward a Logical/Physical Theory of Spreadsheet Modelling. *ACM Trans. Infor. Syst.* **13**(1), 1–37.

Kay, A. (1984) Computer Software. *Sci. Am.* **251**(3), 41–47.

Lerch, J. F., Mantei, M. M. and Olson, J. R. (1989) Skilled Financial Planning: The Cost of Translating Ideas Into Action. *ACM Conf. on Human Factors in Computing Systems*, pp. 121–126.

Lewis, C. and Olson, G. M. (1987) Can Principles of Cognition Lower the Barriers to Programming? *2nd Workshop on Empirical Studies of Programmers*, pp. 248–263.

Lisper, B. and Malmström, J. (2002) Haxcel: A Spreadsheet Interface to Haskell. *14th Int. Workshop on the Implementation of Functional Languages*, pp. 206–222.

Mittermeir, R. and Clermont, M. (2002) Finding High-Level Structures in Spreadsheet Programs. *9th Working Conference on Reverse Engineering*, pp. 221–232.

Norman, D. A. (1986) Cognitive Engineering. In: Norman, D. A. and Draper, S. W. (editors), *User-Centered System Design*, pp. 31–61. Lawrence Erlbaum.

Panko, R. R. (2000) Spreadsheet Errors: What We Know. What We Think We Can Do. *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*.

Peyton Jones, S. L. (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Peyton Jones, S. L., Blackwell, A. and Burnett, M. M. (2003) A User-Centered Approach to Functions in Excel. *ACM Int. Conf. on Functional Programming*, pp. 165–176.

Prabhakarao, S., Cook, C., Ruthruff, J., Creswick, E., Main, M., Durham, M. and Burnett, M. (2003) Strategies and Behaviors of End-User Programmers with Interactive Fault Localization. *IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pp. 203–210.

Rajalingham, K., Chadwick, D. R. and Knight, B. (2001) Classification of Spreadsheet Errors. *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*.

Ronen, B., Palley, M. A. and Lucas, Jr., H. C. (1989) Spreadsheet Analysis and Design. *Comm. ACM*, **32**(1), 84–93.

Rothermel, G., Burnett, M. M., Li, L., DuPuis, C. and Sheretov, A. (2001) A Methodology for Testing Spreadsheets. *ACM Trans. Softw. Eng. & Methodology*, 110–147.

Ruthruff, J., Creswick, E., Burnett, M. M., Cook, C., Prabhakararao, S., Fisher II, M. and Main, M. (2003) End-User Software Visualizations for Fault Localization. *ACM Symp. on Software Visualization*, pp. 123–132.

Sajaniemi, J. (2000) Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *J. Visual Lang. & Comput.* **11**, 49–82.

U.S. Department of Education (2003) *Audit of the Colorado Student Loan Program's Establishment and Use of Federal and Operating Funds for the Federal Family Education Loan Program*. Report ED-OIG/A07-C0009.

U.S. Department of Health and Human Services (2003) *Review of Medicare Bad Debts at Pitt County Memorial Hospital*. Report A-04-02-02016.

Yoder, A. G. and Cohn, D. L. (1994) Real Spreadsheets for Real Programmers. *Int. Conf. on Computer Languages*, pp. 20–30.