Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection

Tie Cheng[1,2,3] and Xavier Rival[1,2]

[1] École Normale Supérieure, Paris, France
[2] INRIA Paris–Rocquencourt, France
[3] École Polytechnique, Palaiseau, France
{tie.cheng,xavier.rival}@ens.fr

Abstract. Spreadsheets are widely used, yet they are error-prone. In particular, they feature a weak type system, which allows certain operations that will silently return unexpected results, like comparisons of integer values with string values. However, discovering these issues is hard, since data and formulas can be dynamically set, read or modified.
We propose a static analysis that detects all run-time type-unsafe operations in spreadsheets. It is based on an abstract interpretation of spreadsheet applications, including spreadsheet tables, global re-evaluation and associated programs.
Our implementation supports the features commonly found in real-world spreadsheets. We ran our analyzer on the EUSES Spreadsheet Corpus. This evaluation shows that our tool is able to automatically verify a large number of real spreadsheets, runs in a reasonable time and discovers complex bugs that are difficult to detect by code review or by testing.

1 Introduction

Spreadsheet applications are ubiquitous in areas such as engineering, statistics, finance and management. They combine a flexible tabular representation of data in two-dimensional tables mixing formulas and values with associated programs. For instance, Microsoft Excel includes a version of Visual Basic for Applications (VBA), whereas Open Office has a BASIC language and Google Spreadsheets have Google Apps Script.

Unfortunately, spreadsheet applications are subject to numerous defects, and often produce incorrect results that do not match user understanding as shown in studies like [21, 22]. As an example, the Task Force Report [1] has recently quoted losses of billions of dollars due to errors in spreadsheet applications used in JP-Morgan's Chief Investment Office. More generally, spreadsheet defects may cause the release of wrong information, the loss of money or the taking of wrong decisions. Due to the risks they incur, spreadsheet applications have been attracting increasing attention from spreadsheet users, IT professionals and computer scientists. Indeed, several approaches have been proposed. Among these proposals are new languages [7] and enhancements to functional features of spreadsheets with better language design [19, 26] and implementation [24, 25]; model-driven

engineering environments to allow only safe updates [14]; and studies to detect code smells that indicate weak points in spreadsheets [13, 17]. Additionally, type systems could be built [3, 9, 4–6] to capture value meanings such as physical units (e.g., apples, oranges) or dimensions (e.g., meters, kilometers) and to verify the correctness of formulas.

While the existing work focuses on spreadsheet tables at the interface level, the other important component of spreadsheet applications, i.e., associated programs (e.g., VBA programs) behind the interface, is seldom addressed. Associated programs can have a significant impact on spreadsheet contents either through a function call from a spreadsheet formula or through an execution of a subroutine launched by users, and are massively present in industrial spreadsheet applications.

Target of the Verification. To support the development of spreadsheet formulas and associated programs, spreadsheet languages supply basic operators as well as many practical built-in functions such as match functions, text functions, logical functions, and date and time functions. Although basic operators and functions have expectations or requirements for the types of their arguments, the weak type system of spreadsheet languages does not often consider a type mismatch as an error, even though meaningless, unexpected or incorrect results may be silently produced. Taking as an example the implicit conversion that spreadsheet languages permit, Microsoft Excel converts the empty value to **true** in expression $\texttt{AND}(\epsilon, \textbf{true})$, whereas it converts it to **false** in expression $\texttt{IF}(\epsilon, 1, 0)$. It will also evaluate comparison $\text{“”} < n$ to **false**, yet the empty string does not have an obvious numeric value. More generally, a mismatch of types is possible, and even likely in practice, but it does not necessarily return an obvious error message (e.g., #VALUE!), and very rarely does it block the execution of spreadsheet applications. Thus, users develop and run spreadsheet applications in the environment where program defects are hidden and yet would be caught by a stricter type system. Consequently, to verify that a spreadsheet application is exempt of any defect of this kind, a *stronger type discipline* is required, and *strong typing information* about formula operands should be inferred.

Static Analysis of Spreadsheets, including Macros by Types. The existing proposals verify mainly the current state of spreadsheet contents, they assume the data in spreadsheets are fully evaluated, and they do not assess the correctness of all the instances of a spreadsheet with different input data. However, many spreadsheets are implemented as applications that handle non-deterministic or non-statically known input in different scenarios: (1) input data may be left blank in the first place and entered later after the formulas have been built (this scenario can also be formally supported by the “Data Validation” feature of Microsoft Excel, which is used to define restrictions on what data can or should be entered in an area); (2) data can be modified manually by users or dynamically read (e.g., by non-deterministic functions such as `RAND`, by accessing the web, or by linking to external databases); (3) if data and formulas are edited in non-automatic calculation mode, data may be outdated; (4) data and formulas can be set and manipulated by associated programs.

Therefore, in this paper, we propose a more complete vision of spreadsheet applications: the applications consist of *formulas* in the spreadsheet that are supported by *associated programs* (macros); the applications may receive input data that is *unknown at verification time* (i.e., that will be read at run-time or non-deterministically); their execution consists in *globally evaluating spreadsheet formulas* or *running an associated program*.

To detect all type related run-time defects of these applications in a fully automatic way, we propose a sound static analysis. The analysis relies on a stronger type system and on an abstract domain which ties properties such as types (e.g., string, integer) to *zones* in spreadsheet tables. The inference of invariants proceeds by abstract interpretation of the spreadsheet applications, and the analysis is not complete. The set of type-unsafe operations is a parameter of the analysis, so that users can select which behaviors are deemed unsafe and should be detected. Our analysis either proves the correctness or visualizes the potential errors to developers, and it provides a high-level view and understanding of their program. The analysis has several benefits: (1) it unearths errors that would not emerge in a dynamic test or user test by computing an over-approximation of all the possible states an execution can reach even in the presence of inputs at run-time; (2) the analysis is efficient enough to be undertaken during the development of a spreadsheet application, though performing it outside the development environment has its own use (e.g., to scan a set of applications while they are not being used).

In this paper, we make the following contributions:

- We set up a concrete model for reasoning about spreadsheet applications, which serves as a basis for the definition and the proof of our analysis (Sect. 3);
- We propose an abstraction for spreadsheet applications, embracing an abstraction for spreadsheet formulas, which is adapted to the verification of typing errors (Sect. 4);
- We define a static analysis for spreadsheet applications, which takes into account both the spreadsheet contents and the actions of the associated programs (Sect. 5);
- We introduce several iteration strategies to perform the analysis of global re-evaluation of spreadsheet contents (Sect. 5);
- We discuss various issues that our tool handles to analyze real-world spreadsheets (Sect. 6) and report on results of verification of the EUSES Spreadsheet Corpus produced by our tool (Sect. 7).

2 Overview

In this section, we consider an example taken from a realistic spreadsheet application (Fig. 1), which silently produces wrong results, due to improper use of operators, in the presence of a very weak type system, as found in spreadsheet environments. This application is made up of a *spreadsheet table* shown in Fig. 1(a) and an *associated program* displayed in Fig. 1(b).

The spreadsheet table contains several columns storing asset variations and values expressed in two currencies, and computes the number of weekdays where

	1	2	3	4	5
1	**Day**	**Delta**	**Delta**	**Delta**	**Total**
2		*(cur1)*	*(cur1)*	*(cur 2)*	*(cur 2)*
3					100
4	Mon	-2	-2	$= \mathbf{C}[4,3] * 1.3$	$= \texttt{IF}(\texttt{ISBLANK}(\mathbf{C}[4,3]), \text{“”}, \mathbf{C}[4,4] + \mathbf{C}[3,5])$
⋮					
9	Sat	0	-4	$= \mathbf{C}[9,3] * 1.3$	$= \texttt{IF}(\texttt{ISBLANK}(\mathbf{C}[9,3]), \text{“”}, \mathbf{C}[9,4] + \mathbf{C}[8,5])$
10	Sun	0	5	$= \mathbf{C}[10,3] * 1.3$	$= \texttt{IF}(\texttt{ISBLANK}(\mathbf{C}[10,3]), \text{“”}, \mathbf{C}[10,4] + \mathbf{C}[9,5])$
⋮					
33	Tue	8	20	$= \mathbf{C}[33,3] * 1.3$	$= \texttt{IF}(\texttt{ISBLANK}(\mathbf{C}[33,3]), \text{“”}, \mathbf{C}[33,4] + \mathbf{C}[32,5])$
34	Wed	-3		$= \mathbf{C}[34,3] * 1.3$	$= \texttt{IF}(\texttt{ISBLANK}(\mathbf{C}[34,3]), \text{“”}, \mathbf{C}[34,4] + \mathbf{C}[33,5])$
⋮					
43	Fri	20		$= \mathbf{C}[43,3] * 1.3$	$= \texttt{IF}(\texttt{ISBLANK}(\mathbf{C}[43,3]), \text{“”}, \mathbf{C}[43,4] + \mathbf{C}[42,5])$
44					**Number of days where asset** > 150
45					$= \texttt{SUM}(\texttt{N}(\mathbf{C}[4,5] : \mathbf{C}[43,5] > 150))$

(a) Spreadsheet contents: Columns 1 and 2 are reserved for input data; Columns 4 and 5 contain pre-coded formulas; Column 3 contains intermediate output and $\mathbf{C}[45,5]$ is the final result.

```
1  Sub Macro()
2    INITIATE;
3    Dim i As Int;
4    Dim j As Int;
5    CLEAR_ZONE(4, 3, 43, 3);
6    i = 4;
7    j = 4;
8    While (j < 44)
9      If C[j, 1] <> "Sat"
           And C[j, 1] <> "Sun"
       Then
10       C[i, 3] = C[j, 2];
11       i = i + 1
12     End;
13     j = j + 1
14   End;
15   Eval
16 End
```

(b) Associated program

Fig. 1. Erroneous behaviors in a spreadsheet application

the total value was greater than a given amount. The area with blue borders in Columns 1 and 2 is reserved for input data, which are the names of the days and the variation in values for each weekday (we note that no variation occurs on the weekend). The associated program shown in Fig. 1(b) and the spreadsheet formulas in the area with green borders are pre-coded. The associated program is run, upon user request, to eliminate meaningless empty weekend values of Column 2, and populate the sequential list into Column 3. The formulas in Column 4 convert the variations stored in Column 3 into another currency. Last, the formulas in Column 5 compute the sequence of meaningful variations, and the number of weekdays where the total asset value was greater than 150, in the bottom right cell.

In practice, to run this application, users fill in data in the input area either manually or automatically (e.g., copying from somewhere else, or using another associated program `INITIATE`). Then, they launch the associated program to compute the values in Column 3, which, in turn, forces the re-evaluation of the formulas stored in Columns 4 and 5 using statement **Eval** in Line 15. The input data, their

array size and the currency rate (i.e., 1.3 in the current example) may be known only at run-time, whereas the spreadsheet formulas and the associated program are pre-coded.

The number shown in the bottom right cell is the final result, yet it is **incorrect**. Let us use the convention that $\mathbf{C}[i, j]$ denotes the cell corresponding to row i and column j. In Fig. 1(a), the cells in the region $\mathbf{C}[34, 5] : \mathbf{C}[43, 5]$ evaluate to the empty string, since the cells in $\mathbf{C}[34, 3] : \mathbf{C}[43, 3]$ are empty (Function `ISBLANK` checks if a cell is empty). Because the comparison operator ">" always returns **true** when applied to a string and a numeric value, when cell $\mathbf{C}[i, 5]$ is an empty string, the condition $\mathbf{C}[i, 5] > 150$ evaluates to **true**. Then, Function `N` converts **true** into 1. **As a consequence, the value produced when evaluating the formula in $\mathbf{C}[45, 5]$ is off by 10**.

This wrong result is issued without raising any warning because of the semantics of the language with a very relaxed type discipline. Such issues are quite common in large spreadsheet applications; they are also hard to diagnose, and usually overlooked by non-experienced users. Furthermore, in some cases this problem can be hidden: if the input data is so large that there is no empty cell in $\mathbf{C}[4, 3] : \mathbf{C}[43, 3]$, the comparison of string and numeric value will not occur. This means that the absence of defects in some cases would not guarantee the entire safety of the applications. Checking the absence of defects by testing is simply not possible, especially when input data are made available at run-time. Instead, detecting such issues requires a conservative analysis that raises a warning whenever an unsafe operation might be executed. For instance, it should warn of the comparison of a string with any numeric value in our example. In general, as users may have different safety standards and different levels of spreadsheet programming, the set of unsafe operations should be a *parameter* of the analysis.

Overall Analysis of the Example. Then we need to look at the properties we can retrieve from the application, which is displayed in Fig. 2. Just after Line 14 of the associated program executes, `j` is known to be equal to 44, whereas the value of `i` can only be determined to be in $[4, 44]$ (as $4 \leq \texttt{i} \leq \texttt{j}$). The first diagram shows the abstraction of formulas that can be determined for *zones* in the spreadsheet (formula zones of constant values are omitted). Each rectangle describes a zone, and is labeled with a formula common to all cells in that zone, in relative indexes (e.g. $\mathbf{C}[+0, -1]$) and absolute indexes (e.g. $\mathbf{C}[4, 5]$). The second diagram shows the current abstraction of types for zones, which will be updated by **Eval** of Line 15. In the analysis of **Eval**, the type for zone $\mathcal{Z}_{f0}$ is first inferred. As the multiplication of *empty* value ϵ and a float value produces float value 0.0, as does the multiplication of two float values, the type for zone $\mathcal{Z}_{f0}$ remains **Float**. Then the type for zone $\mathcal{Z}_{f1}$ is inferred, which results in a sub-zone of **Float** and a sub-zone of **String** (let $\mathcal{Z}'_t$ denote it), whose size may be different from the current string zone ($\mathcal{Z}_t$), and may not be empty. Finally the type of cell $\mathbf{C}[45, 5]$ is inferred, which requires the types in Column 5 including zone $\mathcal{Z}'_t$, and thus involves the unsafe comparison of **String** and **Float**.

Moreover, the verification should not reject obviously correct applications. For instance, a modified version of the example application would replace formulas in

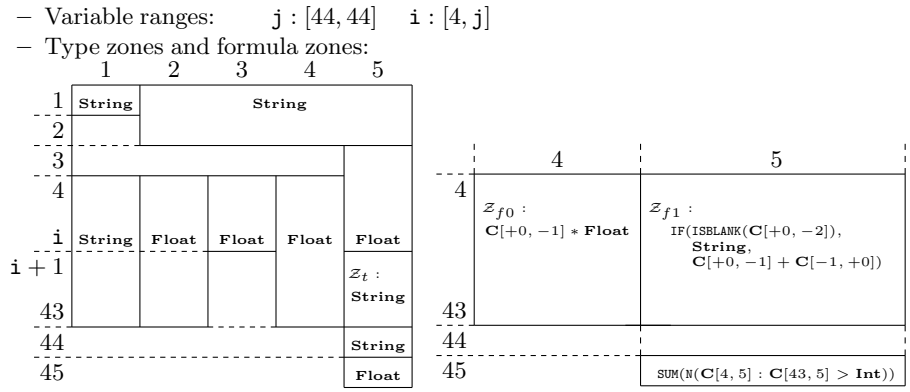


Fig. 2. Abstract state

Column 5 with formulas of the form $\texttt{IF}(\texttt{ISBLANK}(\mathbf{C}[4,3]), 0.0, \mathbf{C}[4,4] + \mathbf{C}[3,5])$. Then, all results in Column 5 would have a floating point type, and no comparison of a string with a numeric value would be performed anymore.

After formalizing a core spreadsheet system, we describe this abstraction formally in Sect. 4, and design an abstract interpretation based analysis able to infer those invariants in Sect. 5.

3 A Core Spreadsheet Language

In this section, we isolate and formalize a core language that incorporates both the spreadsheet table and the runnable code. Note that the analyzer shown in Sections 6 and 7 supports a much wider feature set than this core language. This language has several distinctive characteristics. First, a *spreadsheet application* comprises both the two-dimensional *spreadsheet table* itself (called for short *spreadsheet*) and *associated programs*, which may be run upon user request. Second, a spreadsheet cell contains both *a formula* and *a value* (which is normally displayed to the user). Last, the formulas stored in cells can be *re-evaluated* upon request. Automatic re-evaluation of the whole spreadsheet after cell modification is often deactivated in industrial applications; then, *re-evaluation* can be triggered by a specific command or instruction in the associated program (often used at the end of its execution).

Core Language Syntax. A basic value is either an integer $n \in \mathbb{V}_{\text{int}}$, a floating point $f \in \mathbb{V}_{\text{float}}$ or a string $s \in \mathbb{V}_{\text{string}}$. We write $\mathbb{V} = \mathbb{V}_{\text{int}} \uplus \mathbb{V}_{\text{float}} \uplus \mathbb{V}_{\text{string}} \uplus \{\epsilon, \Omega_{\text{e}}, \Omega_{\text{t}}\}$, where ϵ stands for value "undefined", and where Ω_{e} and Ω_{t} stand for an execution error and a typing error, respectively. We let $\mathbb{X} = \{\mathtt{x}, \mathtt{y}, \ldots\}$ denote a finite set of variables. A variable or a cell content has a type and we assume a set of pre-defined data-types such as not only **Bool**, **Float**, **String**, but also **Date** or **Currency** (which also exist in real spreadsheet languages). Moreover, ϵ is the

$$
\begin{array}{l}
n \in \mathbb{V}_{\text{int}} \quad f \in \mathbb{V}_{\text{float}} \quad s \in \mathbb{V}_{\text{string}} \quad \mathtt{x}, \mathtt{y}, \ldots \in \mathbb{X} \\
v ::= n \mid f \mid s \\
t ::= \textbf{Bool} \mid \textbf{Float} \mid \textbf{Int} \mid \textbf{String} \mid \textbf{Empty} \mid \textbf{Currency} \mid \textbf{Date} \\
e ::= v \quad | \quad \mathtt{x} \quad | \quad \mathbf{C}[e, e] \quad | \quad \mathbf{C}[\pm e, \pm e] \\
\quad | \quad e \oplus e \quad \text{where } \oplus \in \{+, -, \star, \ldots\} \mid \mathtt{F}(e, \ldots, e) \quad \text{where } \mathtt{F} \text{ is a function symbol} \\
s ::= \mathtt{x} = e \quad | \quad \mathbf{C}[e, e] = e \quad | \quad \mathbf{C}[e, e] = \text{“} = e\text{”} \\
\quad | \quad \textbf{Eval} \quad | \quad \textbf{If } e \textbf{ Then } s \textbf{ Else } s \textbf{ End} \quad | \quad \textbf{While}(e)\ s \textbf{ End} \quad | \quad s;\ s \\
d ::= \textbf{Dim } \mathtt{x} \textbf{ As } t \\
a ::= d; \ldots; d; s
\end{array}
$$

Fig. 3. Syntax: a core spreadsheet language

only value of type **Empty**. The spreadsheet itself is a fixed size array of dimension two. Rows (resp., columns) are labeled in a range $\mathbb{R} = \{1, 2, \ldots, n_{\mathbb{R}}\}$ (resp., $\mathbb{C} = \{1, 2, \ldots, n_{\mathbb{C}}\}$). A cell address is referred to in absolute terms, by a pair (i, j) where $i \in \mathbb{R}$ and $j \in \mathbb{C}$.

Expressions appear in both associated programs and spreadsheet formulas. An *expression* $e \in \mathbb{E}$ may be either a constant, the reading of a variable or of a cell, or the result of the application of a binary operator or of a built-in function (such as `ISBLANK`, `IF`, `SUM`, etc.). Cell reads in spreadsheet formulas should correspond to constant indexes, but may be relative to the position of the cell they appear in: for instance, formula $\mathbf{C}[-1, +0]$ in cell $\mathbf{C}[3, 4]$ corresponds to cell $\mathbf{C}[2, 4]$. A statement s may be either a variable declaration (together with its type), or an assignment, or an evaluation statement or a control structure (sequence, condition test, loop). Assignments may modify either the contents of a variable or the contents of a cell. Assignment to a cell may store either an evaluated value as in $\mathbf{C}[e_0, e_1] = e_2$ or a formula and its currently evaluated value as in $\mathbf{C}[e_0, e_1] =$ “ $= e_2$” (in the latter case, it may be re-evaluated in the future). Last, statement **Eval** causes a global re-evaluation of the *whole* spreadsheet (real spreadsheet software typically allows a finer-grained control of re-evaluation, which we do not model here, as its behavior is similar to our global **Eval**).

Spreadsheet Applications. In Excel, a *spreadsheet application* comprises one spreadsheet and a set of associated programs, which may be run either immediately, or upon user request. In the following, and without a loss in generality, we assume that an application is defined by a single application that includes the initialization of the spreadsheet of the form $\mathbf{C}[i, j] = e$ or $\mathbf{C}[i, j] =$ “ $= e$”. Thus, the example of Sect. 2 would actually be represented by a single associated program filling in the spreadsheet with values and formulas prior to the body shown in Fig. 1(b). This allows us to describe many real spreadsheet applications with the core language shown in Fig. 3. Finally, we assume a spreadsheet application a is defined by a sequence of declarations, followed by a program s, which begins with the assignment of all cells that need to be initialized. Moreover, our implementation takes into account many additional features of spreadsheet environments such as data validation or circular references, which will be covered in Sect. 6.

Example 1 (Simple application). The application below declares one variable; it then fills in 4 cells, and modifies one (a global re-evaluation takes place in the middle of the process).

1 **Dim** x **As Int**;	5 **Eval**;
2 $x = -5$;	6 $\mathbf{C}[1,1] = 24$;
3 $\mathbf{C}[1,1] = 6$;	7 $\mathbf{C}[2,2] =$ " $= \mathbf{C}[1,1] + 8$";
4 $\mathbf{C}[2,1] =$ " $= \mathbf{C}[1,1]$";	8 $\mathbf{C}[3,2] =$ " $= \mathbf{C}[2,1] + \mathbf{C}[2,2]$"

Semantic Domains. At any time in the execution, the memory is characterized by the values of variables, and the formulas and values contained in the spreadsheet. Thus, a *non-error concrete state* consists of a 3-tuple $(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}})$ where $\sigma^{\mathbb{X}} \in \mathbb{X} \to \mathbb{V}$ maps each variable to its value, $\sigma^{\mathbb{SE}} \in \mathbb{S}_{\mathbb{E}} = (\mathbb{R} \times \mathbb{C} \to \mathbb{E})$ maps each cell to the formula it contains and $\sigma^{\mathbb{SV}} \in \mathbb{S}_{\mathbb{V}} = (\mathbb{R} \times \mathbb{C} \to \mathbb{V})$ maps each cell to the value it contains. We write Σ for the set of such concrete states. For instance, the evaluation of the application of Example 1 produces the state shown below as a graphical view (we show only the results for cells in the first two columns and the first three rows as the others are empty):

$$\sigma^{\mathbb{X}} : \mathtt{x} \mapsto -5$$

$\sigma^{\mathbb{SE}}$	1	2
1	= 24	
2	= C[1, 1]	= C[1, 1] + 8
3		= C[2, 1] + C[2, 2]

$\sigma^{\mathbb{SV}}$	1	2
1	24	ϵ
2	6	32
3	ϵ	38

Semantics of Expressions. The evaluation of an expression e is defined by its semantic function $[\![e]\!]_{\mathbb{E}} : \Sigma \to \mathcal{P}(\mathbb{V})$ (note that we assume an expression may evaluate to several values in order to account for possible non-determinism and run-time inputs, which may arise due to functions such as `RAND`, `DATE`, or other functions to input real-time data). We let $[\![\oplus]\!] : (\mathcal{P}(\mathbb{V}))^2 \to \mathcal{P}(\mathbb{V})$ denote the concrete mathematical function corresponding to operator $\oplus$, and let $[\![\mathtt{F}]\!] : (\mathcal{P}(\mathbb{V}))^n \to \mathcal{P}(\mathbb{V})$ denote the mathematical function associated with built-in (n-ary) function `F`, we remark that their arguments may also be non-deterministic. Then, $[\![e]\!]_{\mathbb{E}}$ can be defined by induction over the syntax, as shown below:

$$\begin{aligned}
[\![v]\!]_{\mathbb{E}}(\sigma) &= \{v\} \quad [\![\mathtt{x}]\!]_{\mathbb{E}}(\sigma) = \{\sigma^{\mathbb{X}}(\mathtt{x})\} \\
[\![\mathbf{C}[e_0, e_1]]\!]_{\mathbb{E}}(\sigma) &= \{\sigma^{\mathbb{SV}}(v_0, v_1) \mid \forall i,\ v_i \in [\![e_i]\!]_{\mathbb{E}}(\sigma)\} \\
[\![e_0 \oplus e_1]\!]_{\mathbb{E}}(\sigma) &= \bigcup\{[\![\oplus]\!](v_0, v_1) \mid \forall i,\ v_i \in [\![e_i]\!]_{\mathbb{E}}(\sigma)\} \\
[\![\mathtt{F}(e_1, \ldots, e_n)]\!]_{\mathbb{E}}(\sigma) &= \bigcup\{[\![\mathtt{F}]\!](v_1, \ldots, v_n) \mid \forall i,\ v_i \in [\![e_i]\!]_{\mathbb{E}}(\sigma)\}
\end{aligned}$$

We remark that this evaluation function uses the last evaluated value whenever it reads a cell. In particular, it does not evaluate the cells it reads the value of, nor their ancestors. Therefore, (1) an update of an ancestor of a cell c will not cause the update of the value in c, which means the value in c may become "outdated"; (2) when a cell value is outdated, any evaluation function that uses its value returns a possibly outdated result. For instance, in Example 1, after the

global re-evaluation in Line 5, $\sigma^{\mathbb{SV}}(2,1) = 6$, since $\sigma^{\mathbb{SV}}(1,1) = 6$. In Line 6, the value of $\mathbf{C}[1,1]$ changes, then $\mathbf{C}[2,1]$ as its descendant becomes outdated. In Line 8, $[\![\mathbf{C}[2,1] + \mathbf{C}[2,2]]\!]_{\mathbb{E}}(\sigma) = \{38\}$ is calculated from the outdated value of $[\![\mathbf{C}[2,1]]\!]_{\mathbb{E}}(\sigma) = \{6\}$; thus, $\mathbf{C}[3,2]$ is outdated too.

Errors. The evaluation of some expressions may fail to produce a value. A common case is division by 0, or a cell read with invalid (e.g., negative) row and column indexes. These errors, represented by Ω_{e}, are treated by other techniques and are not studied in this paper. Whereas, we are interested in typing errors which we note Ω_{t} and which may arise when applying an operator / function to arguments whose types do not match the convention or the expectation of that operator / function. For instance, as the comparison between a floating point value and a string is considered unsafe, we have $\forall v_{\mathrm{f}} \in \mathbb{V}_{\mathrm{float}},\ \forall v_{\mathrm{s}} \in \mathbb{V}_{\mathrm{string}},\ [\![>]\!](v_{\mathrm{f}}, v_{\mathrm{s}}) = \Omega_{\mathrm{t}}$. Moreover, as a value, Ω_{t} has no type. With a slight abuse of notation, we also let Ω_{t} denote the abstract typing error value, and the blocking error state which contains any typing error value in both concrete and abstract semantics.

Semantics of Program Statements. The concrete semantics of a statement, program, or program fragment s is a function mapping an initial state to the set of final states that can be reached after executing it: $[\![s]\!]_{\mathbb{P}} : \Sigma \to \mathcal{P}(\Sigma)$. It can also be computed by induction over the syntax. For instance:

- $[\![s_0;\ s_1]\!]_{\mathbb{P}}(\sigma) = \bigcup\{[\![s_1]\!]_{\mathbb{P}}(\sigma_0) \mid \sigma_0 \in [\![s_0]\!]_{\mathbb{P}}(\sigma)\}$;
- $[\![\textbf{If } e \textbf{ Then } s_0 \textbf{ Else } s_1 \textbf{ End}]\!]_{\mathbb{P}}(\sigma) = S_0 \cup S_1$ where $S_0 = [\![s_0]\!]_{\mathbb{P}}(\sigma)$ if $\textbf{true} \in [\![e]\!]_{\mathbb{E}}(\sigma)$ and $S_0 = \emptyset$ otherwise (and the same for S_1, w.r.t. the second branch);
- as usual, the semantics of a loop involves a least-fixpoint computation.

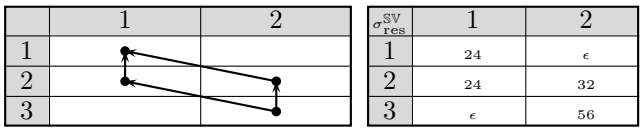
All forms of assignment statements (to a variable or to a cell) always trigger immediate evaluation. The semantics of assignment to a variable is straightforward: $[\![\mathtt{x} = e]\!]_{\mathbb{P}}(\sigma) = \{(\sigma^{\mathbb{X}}[\mathtt{x} \leftarrow v], \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}) \mid v \in [\![e]\!]_{\mathbb{E}}(\sigma)\}$; the only difference between the two other assignments is that the formula in the right-hand side is preserved only in the latter case:

- Assignment of a value to a cell: $[\![\mathbf{C}[e_0, e_1] = e_2]\!]_{\mathbb{P}}(\sigma) = \{(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}[(v_0, v_1) \leftarrow v_2], \sigma^{\mathbb{SV}}[(v_0, v_1) \leftarrow v_2]) \mid \forall i \in \{0,1,2\},\ v_i \in [\![e_i]\!]_{\mathbb{E}}(\sigma)\}$
- Assignment of a formula: $[\![\mathbf{C}[e_0, e_1] = \text{“ } = e_2\text{”}]\!]_{\mathbb{P}}(\sigma) = \{(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}[(v_0, v_1) \leftarrow e_2], \sigma^{\mathbb{SV}}[(v_0, v_1) \leftarrow v_2]) \mid \forall i \in \{0,1,2\},\ v_i \in [\![e_i]\!]_{\mathbb{E}}(\sigma)\}$

Semantics of Global Spreadsheet Re-evaluation. The evaluation statement causes all cells in the whole spreadsheet to be re-evaluated. Therefore, the semantics of **Eval** involves a possibly large number of computation steps, where the value of each cell is recomputed. This naturally boils down to a *fixpoint* computation over the whole spreadsheet, where $\sigma^{\mathbb{SV}}$ is recalculated.

In this section, we consider spreadsheet environments without circular references (which will be covered in Sect. 6). Any such spreadsheet has an acyclic cell dependency graph. By following a topological ordering of the cells, the formulas contained in cells are evaluated one by one. For instance, if we consider the state shown in Example 1, the dependencies are shown in the left figure below. Therefore, if we only take into account non-empty cells, total orderings

$(1,1) \prec (2,1) \prec (2,2) \prec (3,2)$ and $(1,1) \prec (2,2) \prec (2,1) \prec (3,2)$ can be used for the computation (a non-total ordering could also be considered). Re-evaluation using any of these orders produces the state $(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}_{\text{res}})$ where $\sigma^{\mathbb{SV}}_{\text{res}}$ is on the right:



	1	2
1		
2		
3		

$\sigma^{\mathbb{SV}}_{\text{res}}$	1	2
1	24	ϵ
2	24	32
3	ϵ	56

As we intend to perform an abstract interpretation based static analysis of programs, and since abstract interpretation relies on *fixpoint transfer* theorems to derive sound analyses from the concrete semantics, we now formalize the definition of the semantics of **Eval** as a least-fixpoint. Following the intuitive calculation scheme defined above, we can define $[\![\mathbf{Eval}]\!]_{\mathbb{P}}(\sigma)$ as a fixpoint where each iterate computes exactly one cell.

In the following, we let $\prec$ denote a topological ordering over $\mathbb{R} \times \mathbb{C}$. A computation step calculates the lowest cell in ordering $\prec$, that has not been computed yet, and that can be computed. To distinguish cells whose value has been computed from cells that remain to be re-evaluated, we introduce an additional $\bot$ value. We formalize this notion of computation step with a binary relation $\rightsquigarrow_{\prec}$, which is such that $\sigma^{\mathbb{SV}}_0 \rightsquigarrow_{\prec} \sigma^{\mathbb{SV}}_1$ if and only if: $\sigma^{\mathbb{SV}}_1(i,j) \in [\![\sigma^{\mathbb{SE}}(i,j)]\!]_{\mathbb{E}}(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}_0)$ when $\sigma^{\mathbb{SV}}_0(i,j) = \bot$ and $\forall (i',j') \prec (i,j),\ \sigma^{\mathbb{SV}}_0(i',j') \neq \bot$; otherwise $\sigma^{\mathbb{SV}}_1(i,j) = \sigma^{\mathbb{SV}}_0(i,j)$. We remark that $\sigma^{\mathbb{SV}} \rightsquigarrow_{\prec} \sigma^{\mathbb{SV}}$, when $\sigma^{\mathbb{SV}}$ is fully computed (i.e., when no unevaluated formula remains). Then, the iteration function $\mathcal{F}_{\prec} : \mathcal{P}(\mathbb{S}_{\mathbb{V}}) \to \mathcal{P}(\mathbb{S}_{\mathbb{V}})$ is defined as $\mathcal{F}_{\prec}(\mathcal{S}) = \{\sigma^{\mathbb{SV}}_1 \in \mathbb{S}_{\mathbb{V}} \mid \exists \sigma^{\mathbb{SV}}_0 \in \mathcal{S},\ \sigma^{\mathbb{SV}}_0 \rightsquigarrow_{\prec} \sigma^{\mathbb{SV}}_1\}$.

We now need to set up a lattice structure where the computation of the least-fixpoint should take place. As the computation progresses by filling in more cells, we need an order relation over spreadsheets which captures property "$\sigma^{\mathbb{SV}}_1$ has more evaluated cells than $\sigma^{\mathbb{SV}}_0$ and they agree on common evaluated cells", allowing for the value of a cell to move from $\bot$ to any other value. First, we let $\sqsubseteq_{\mathbb{V}}$ be the relation over the set of values extended by a constant $\top$ (denoting the definition contradiction) defined by the lattice: $\forall v \in \{\ldots, -1, 0, 1, \ldots, \epsilon, \mathbf{true}, \mathbf{false}, \ldots\},\ \bot \sqsubseteq_{\mathbb{V}} v \sqsubseteq_{\mathbb{V}} \top$. This relation extends to sets of spreadsheets: $\forall \mathcal{S}_0, \mathcal{S}_1 \in \mathcal{P}(\mathbb{S}_{\mathbb{V}}),\ \mathcal{S}_0 \sqsubseteq \mathcal{S}_1$ if and only if $\forall \sigma^{\mathbb{SV}}_0 \in \mathcal{S}_0,\ \exists \sigma^{\mathbb{SV}}_1 \in \mathcal{S}_1,\ \forall (i,j) \in \mathbb{R} \times \mathbb{C},\ \sigma^{\mathbb{SV}}_0(i,j) \sqsubseteq_{\mathbb{V}} \sigma^{\mathbb{SV}}_1(i,j)$. Moreover, we let $\sigma^{\mathbb{SV}}_{\bot} \in \mathbb{S}_{\mathbb{V}}$ be defined by $\forall (i,j),\ \sigma^{\mathbb{SV}}_{\bot}(i,j) = \bot$. At this stage, we can define the semantics of the re-evaluation as the fixpoint of $\mathcal{F}_{\prec}$:

Theorem 1 (Definition of $[\![\mathbf{Eval}]\!]_{\mathbb{P}}$). *For all pairs of topological orders $\prec, \prec'$ compatible with the dependencies induced by formulas stored in cells ($\sigma^{\mathbb{SE}}$), we have:* $\mathbf{lfp}_{\sigma^{\mathbb{SV}}_{\bot}} \mathcal{F}_{\prec} = \mathbf{lfp}_{\sigma^{\mathbb{SV}}_{\bot}} \mathcal{F}_{\prec'} = \bigsqcup \{(\mathcal{F}_{\prec})^n(\sigma^{\mathbb{SV}}_{\bot}) \mid n \in \mathbb{N}\}$. *Thus, we define:* $[\![\mathbf{Eval}]\!]_{\mathbb{P}}(\sigma) = \{(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}_{\text{res}}) \mid \sigma^{\mathbb{SV}}_{\text{res}} \in \mathbf{lfp}_{\sigma^{\mathbb{SV}}_{\bot}} \mathcal{F}_{\prec}\}$

Another property that follows from the absence of circular dependencies is the fact that value $\top$ never arises in the spreadsheets obtained in the set defined by this fixpoint. Moreover, all values are defined (i.e., not equal to $\bot$) and empty cells (with no formula) contain value ϵ. We can also remark that the least fixpoint is obtained after at most $n_{\mathbb{R}} \cdot n_{\mathbb{C}}$ iterations. Spreadsheet environments typically use

a *total* topological order, in order to obtain a sequential computation of the fixpoint. This is not mandatory in Theorem 1, and this definition permits to perform "parallel computation" (i.e., in the same iterate) of cells that can be defined in the same time (but each cell is computed exactly once).

Semantics of a Spreadsheet Application. In order to reason about safety properties for a spreadsheet application a, we need to set up a semantics $[\![a]\!]_{\mathbb{A}} \subseteq \Sigma$ which collects *all* the states (not only final states, as $[\![s]\!]_{\mathbb{P}}$ does) that can be reached at any point in the execution of the application. The full definition of $[\![a]\!]_{\mathbb{A}}$ follows from that of $[\![s]\!]_{\mathbb{P}}$ and is based on a trivial fixpoint, starting from the initial state σ_{i} where all variables, formulas, and values are set to ϵ.

4 Abstraction

We now formalize the *abstraction* [10] used in our analysis (Sect. 5), which is based on *abstract formulas* (Sect. 4.1) which summarize the behavior of formulas depending on the type of their inputs and on *abstract zones* [8] which permit to tie abstract predicates to sets of spreadsheet cells (Sect. 4.2).

4.1 Formula Abstraction

The computation of type information over zones requires the propagation of information not only through the associated program, but also through the formulas contained in the spreadsheet itself, to be able to analyze re-evaluation. Therefore, the effect of formulas should be propagated through the analysis. However, dealing with all formulas contained in the spreadsheet would be too costly. Therefore, we propose an abstraction of the effect of formulas, which expresses their effect on types, and replaces, e.g., constants with a type:

Definition 1 (Abstract formulas). *The set* $\mathbb{E}^\sharp$ *of* abstract formulas *is defined in the grammar below:*

$$\begin{array}{l} n \in \mathbb{V}_{\mathrm{int}}, \quad e^\sharp \in \mathbb{E}^\sharp, \quad t \in \mathbb{T} \\ e^\sharp ::= t \mid \mathbf{C}[n,n] \mid e^\sharp \oplus e^\sharp \mid \mathtt{F}(e^\sharp,\ldots,e^\sharp) \end{array}$$

Example 2 (Abstract formulas). For instance, $\mathbf{Int}{+}\mathbf{Float}$, $\mathbf{Float}{-}\mathbf{C}[3,4]$, $\mathtt{ISBLANK}(\mathbf{C}[5,6])$ are all abstract formulas. More generally, it is also possible to extend this set of abstract formulas with relative indexes: $\mathbf{C}[{+}0,{-}1]\star\mathbf{Float}$ in Fig. 2 is also a valid abstract formula.

Semantics of Abstract Formulas. Intuitively, we can give a semantics to abstract formulas, following a similar scheme as in Sect. 3, except that an abstract formula evaluates into a *type*. In the following, we let a *type spreadsheet* be a function $\sigma^{\mathbb{T}} \in \mathbb{S}_{\mathbb{T}} = (\mathbb{R}\times\mathbb{C} \to \mathbb{T})$ mapping each cell to a type, and we say spreadsheet contents $\sigma^{\mathbb{SV}}$ has type $\sigma^{\mathbb{T}}$ (noted $\sigma^{\mathbb{SV}} : \sigma^{\mathbb{T}}$) if and only if $\forall (i,j) \in \mathbb{R}\times\mathbb{C},\ \sigma^{\mathbb{SV}}(i,j) : \sigma^{\mathbb{T}}(i,j)$. To define the semantics of abstract formulas, we let each operator $\oplus$ (resp.,

built-in function $\mathtt{F}$) be abstracted by a partial function $[\![\oplus]\!]_{\mathrm{t}} : (\mathcal{P}(\mathbb{T}))^2 \to (\mathcal{P}(\mathbb{T}))$ (resp., $[\![\mathtt{F}]\!]_{\mathrm{t}} : (\mathcal{P}(\mathbb{T}))^n \to \mathcal{P}(\mathbb{T})$) that over-approximates its effect on types. For instance, $[\![+]\!]_{\mathrm{t}}(\{\mathbf{Int}\}, \{\mathbf{Int}\}) = \{\mathbf{Int}\}$ and $[\![*]\!]_{\mathrm{t}}(\{\mathbf{Int}\}, \{\mathbf{Float}\}) = \{\mathbf{Float}\}$. On the other hand, as noted in Sect. 2, it is unsafe to compare a string with an integer, so $[\![<]\!]_{\mathrm{t}}(\{\mathbf{String}\}, \{\mathbf{Int}\})$ leads to Ω_{t}, so are the other unsafe operations. Then, the semantics of abstract formula $e^\sharp$ is a function $[\![e^\sharp]\!]_{\mathbb{T}} : \mathbb{S}_{\mathbb{T}} \to \mathcal{P}(\mathbb{T})$ mapping spreadsheets into sets of types.

Abstraction of Formulas. A spreadsheet formula can be translated into an abstract formula by replacing, e.g., all constants with types, this process is formalized in the definition of the translation function ϕ below:

$$\phi(v) = t \text{ where } t \text{ is the type of } v \qquad \phi(\mathbf{C}[i,j]) = \mathbf{C}[i,j]$$
$$\phi(e_0 \oplus e_1) = \phi(e_0) \oplus \phi(e_1) \qquad \phi(\mathtt{F}(e_1, \ldots, e_n)) = \mathtt{F}(\phi(e_1), \ldots, \phi(e_n))$$

Note that the translation applies only to formulas found in the spreadsheet (i.e. not to general expressions found in associated programs), thus ϕ is not defined for variables or cell accesses of the form $\mathbf{C}[e_0, e_1]$ where e_0 or e_1 is not a constant. The intended effect on types is preserved by ϕ, it satisfies the soundness condition: Let $e \in \mathbb{E}$, $\sigma^{\mathbb{SV}} \in \mathbb{S}_{\mathbb{V}}$ and $\sigma^{\mathbb{T}} \in \mathbb{S}_{\mathbb{T}}$, such that $\forall (i,j),\ \sigma^{\mathbb{SV}}(i,j) : \sigma^{\mathbb{T}}(i,j)$. Then, $\forall v \in [\![e]\!]_{\mathbb{E}}(\sigma),\ \exists t \in [\![\phi(e)]\!]_{\mathbb{T}}(\sigma^{\mathbb{T}}),\ v : t$.

Example 3 (Abstraction of formulas). For instance, we have the abstractions below, taken from the example of Sect. 2:

- $\phi(\mathbf{C}[4,4] * 1.3) = \mathbf{C}[4,3] * \mathbf{Float}$, or if we use relative indexes, $\phi(\mathbf{C}[+0,-1] * 1.3) = \mathbf{C}[+0,-1] * \mathbf{Float}$.

Simplification of Abstract Formulas. Some type formulas may be simplified, while still carrying the same amount of information. For instance, the addition of two floating point values produces a new floating point value, thus type formula $\mathbf{Float} + \mathbf{Float}$ can be simplified into $\mathbf{Float}$. The concrete semantics of functions may allow for less trivial formula simplifications. For example, the function `ISERROR` checks if a value is of type $\mathbf{Error}$; it always returns a boolean value whatever the argument is, then formula $\mathtt{ISERROR}(\mathbf{C}[5,6])$ can be simplified into $\mathbf{Bool}$.

Therefore, a *simplification function* $\mathbf{S} : \mathbb{E}^\sharp \to \mathbb{E}^\sharp$ is usually defined by structural induction over formulas, by applying a set of local rules. It is sound with respect to the concrete semantics: $\forall e^\sharp \in \mathbb{E}^\sharp,\ [\![\mathbf{S}(e^\sharp)]\!]_{\mathbb{T}} = [\![e^\sharp]\!]_{\mathbb{T}}$. Also, an unsafe operation should not be simplified (e.g., simplification rule $\mathbf{S}(\mathbf{Float} > \mathbf{String}) = \mathbf{Bool}$ is not admissible), as these operations are exactly what our analysis aims at finding out.

4.2 Spreadsheet Abstraction

Spreadsheet Zones Abstraction. To abstract spreadsheets, we need to tie abstract properties such as types or abstract formulas to table *zones*. In the following, we use the zone abstraction of [8], where a zone describes a set of cells in a compact manner. A zone abstraction is defined by a numeric abstract domain [10] $\mathbb{D}^\sharp_{\mathrm{num}}$ over

$\mathbb{X}$ (where $\mathbb{X}$ contains two special variable names $\bar{i}$ and $\bar{j}$ that cannot be used in the associated programs and that respectively denote the row and column of a cell), with a concretization function $\gamma_{\text{num}} : \mathbb{D}^\sharp_{\text{num}} \to \mathcal{P}(\mathbb{X} \to \mathbb{V})$. A set of cell coordinates S and a concrete state $(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}})$ is abstracted by zone $\mathcal{Z} \in \mathbb{D}^\sharp_{\text{num}}$ if and only if $\forall (i,j) \in S,\ [\sigma^{\mathbb{X}}, \bar{i} = i, \bar{j} = j] \in \gamma_{\text{num}}(\mathcal{Z})$, i.e., the coordinates in S together with $\sigma^{\mathbb{X}}$ satisfy $\mathcal{Z}$. When not considering an associated program, no other variable than $\bar{i}, \bar{j}$ should appear in $\mathcal{Z}$. In this paper, we employ a variant of difference bound matrices (DBMs) which was used in [8], and inspired from the octagon abstract domain [20].

Example 4 (Abstract zones). 3 zones can be represented by $\mathcal{Z}_0 : \bar{j} = 2 \wedge 4 \leq \bar{i} \wedge \bar{i} \leq 43$, $\mathcal{Z}_1 : \bar{j} = 3 \wedge 4 \leq \bar{i} \wedge \bar{i} \leq \mathtt{i}$ and $\mathcal{Z}_2 : \bar{j} = 4 \wedge 4 \leq \bar{i} \wedge \bar{i} \leq 43$. So $\mathcal{Z}_0$ denotes a block of cells in Column 2 from Row 4 till Row 43. Similarly $\mathcal{Z}_2$ describes a block in Column 4. On the other hand, $\mathcal{Z}_1$ denotes a block in Column 3, from row 4 till Row $n_{\mathtt{i}}$ where $n_{\mathtt{i}}$ denotes the value of $\mathtt{i}$ in the current state (it is thus meaningful only in an associated program).

We abbreviate pairs of bounds on $\mathtt{i}, \mathtt{j}$ using interval notation, and let, e.g., $4 \leq \bar{i} \wedge \bar{i} \leq \mathtt{i}$ denote $\bar{i} \in [4, \mathtt{i}]$ ($[4, \mathtt{i}]$ does not necessarily imply $4 \leq \mathtt{i}$), therefore $\mathcal{Z}_1 : \bar{j} = 3 \wedge \bar{i} \in [4, \mathtt{i}]$. Also, for each coordinate ($\bar{i}$ or $\bar{j}$), if there is only one interval constraining it, we can write a zone as a rectangle. Thus $[4, \mathtt{i}] \times [3, 3]$ and even $[4, \mathtt{i}] \times 3$ are both valid forms to describe $\mathcal{Z}_1$.

State Abstraction. An abstract state encloses (i) numerical abstract properties of variables and (ii) a collection of abstract zone predicates, that is, abstract predicates that hold true over all cells that can be characterized by an abstract zone.

An *abstract predicate* is either a type or an abstract formula. This defines an abstract domain $\mathbb{D}^\sharp_{\text{c}} = \{\bot, \top\} \cup \mathbb{T} \cup \mathbb{E}^\sharp$. To distinguish an abstract type formula from a type, we insert "=" before the type (e.g., **String** $\in \mathbb{T}$, whereas " = **String**" $\in \mathbb{E}^\sharp$). Concretization function $\gamma_{\text{c}} : \mathbb{D}^\sharp_{\text{c}} \to \mathcal{P}(\mathbb{E} \times \mathbb{V})$ is defined by:

$$\forall t \in \mathbb{T},\ \ \gamma_{\text{c}}(t) = \{(e, v) \in \mathbb{E} \times \mathbb{V} \mid v : t\}$$
$$\forall e^\sharp \in \mathbb{E}^\sharp,\ \gamma_{\text{c}}(e^\sharp) = \{(e, v) \in \mathbb{E} \times \mathbb{V} \mid \phi(e) = e^\sharp\}$$

We can now define abstract states as follows:

Definition 2 (Abstract zone predicate and abstract state). *An* abstract zone predicate *is a pair* $(\mathcal{Z}, \mathcal{P})$ *where* $\mathcal{Z} \in \mathbb{D}^\sharp_{\text{num}}$ *and* $\mathcal{P} \in \mathbb{D}^\sharp_{\text{c}}$. *We let* $\mathbb{D}^\sharp_{\text{z}} = \mathbb{D}^\sharp_{\text{num}} \times \mathbb{D}^\sharp_{\text{c}}$. *The concretization* $\gamma_{\text{z}} : \mathbb{D}^\sharp_{\text{z}} \longrightarrow \mathcal{P}(\Sigma)$ *is such that* $(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}) \in \gamma_{\text{z}}(\mathcal{Z}, \mathcal{P})$ *if and only if:*

$$\forall (i,j) \in \mathbb{R} \times \mathbb{C},\ [\sigma^{\mathbb{X}}, \bar{i} = i, \bar{j} = j] \in \gamma_{\text{num}}(\mathcal{Z}) \Longrightarrow (\sigma^{\mathbb{SE}}(i,j), \sigma^{\mathbb{SV}}(i,j)) \in \gamma_{\text{c}}(\mathcal{P})$$

An abstract state *is a pair* $(N^\sharp, P^\sharp) \in \mathbb{D}^\sharp_\Sigma = \mathbb{D}^\sharp_{\text{num}} \times \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z}})$. *Moreover, concretization function* $\gamma_\Sigma : \mathbb{D}^\sharp_\Sigma \to \mathcal{P}(\Sigma)$ *is defined by:*

$$\gamma_\Sigma(N^\sharp, P^\sharp) = \{(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}) \mid \sigma^{\mathbb{X}} \in \gamma_{\text{num}}(N^\sharp) \wedge (\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}) \in \bigcap_{p^\sharp \in P^\sharp} \gamma_{\text{z}}(p^\sharp)\}$$

We may also want to distinguish two kinds of predicates with $P^\sharp$, by letting $F^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,form}})$ denote the abstract zone predicates for formulas and letting $T^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,type}})$ denote those for types. Thus, $P^\sharp = F^\sharp \uplus T^\sharp$, and $(N^\sharp, P^\sharp)$ is equivalent to $(N^\sharp, F^\sharp \uplus T^\sharp)$. The construction of Definition 2 utilizes the *reduced product* and *reduced cardinal power* of abstract domains [11]. It also extends the domain shown in [8] with abstract formulas.

Example 5 (Abstract predicates over spreadsheet zones). Using the same notations as in Example 4, we have the following abstract zone predicates, which are all satisfied in the concrete state of Fig. 1(a):

- Zones $\mathcal{Z}_0$ and $\mathcal{Z}_1$ correspond to values of type **Float**, which are described by the predicates $(\mathcal{Z}_0, \mathbf{Float})$ and $(\mathcal{Z}_1, \mathbf{Float})$.
- All cells in zone $\mathcal{Z}_2$ contain a formula which can be abstracted by $\mathbf{C}[+0, -1] * \mathbf{Float}$, thus we can use abstract zone predicate $(\mathcal{Z}_2, \mathbf{C}[+0, -1] * \mathbf{Float})$ to describe it.

Likewise, Fig. 2 displays an abstract state made of ten zones bound to types and three zones bound to abstract formulas.

5 Static Analysis Algorithms

We now set up a fully automatic static analysis, which computes abstract invariants in the abstract domain defined in Sect. 4. The analysis aims at computing an *over*-approximation of the set $[\![a]\!]_{\mathbb{A}}$ of the reachable concrete states of an application a. It proceeds by *abstract interpretation* [10] of the body of a: the effect of each statement is over-approximated in a sound manner by some adequate transfer functions, and a widening operator enforces the convergence of abstract iterates whenever a concrete fixpoint needs to be approximated in the abstract level. In this section, we design an *abstract semantics* for statements and applications: (1) the abstract semantics $[\![s]\!]^\sharp_{\mathbb{P}} : \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$ of statement s is a function which maps an abstract pre-condition into a conservative abstract post-condition (which is described by abstract states); (2) the abstract semantics $[\![a]\!]^\sharp_{\mathbb{A}} \subseteq \mathbb{D}^\sharp_\Sigma$ of application a is a finite set of abstract states. Both of them should be sound. Before we set up the abstract semantics of common program statements in Sect. 5.2, we define the abstract domain operations in Sect. 5.1, using the abstract domain introduced in [8] as a basis. Section 5.3 describes the abstract interpretation of global re-evaluation statement **Eval**, which is the most involved part of our analysis.

5.1 Abstract Operators over Zone Predicates.

The abstract operations defined in this section are derived from those proposed in [8] to compute abstract zones. Therefore, we describe the structure of abstract operations and refer the reader to [8] for their full description.

Assignment. As mentioned in Sect. 3, our language features three forms of assignments: assignment to a variable is the most simple form, both forms of assignments to a spreadsheet cell are similar. We need to define three assignment transfer functions ($\mathbf{assign}^\sharp_{\mathbb{X}}, \mathbf{assign}^\sharp_{\mathbb{V}}, \mathbf{assign}^\sharp_{\mathbb{E}}$) that share the same principles, thus we focus on formula assignment $\mathbf{assign}^\sharp_{\mathbb{E}} : \mathbb{E} \times \mathbb{E} \times \mathbb{E} \times \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$. Given e_0, e_1, e_2, it should satisfy:

$$\forall \sigma^\sharp \in \mathbb{D}^\sharp_\Sigma,\ \forall (\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}) \in \gamma_\Sigma(\sigma^\sharp), \forall v_{i\in\{0,1,2\}} \in [\![e_i]\!]_{\mathbb{E}}(\sigma),$$
$$(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}[(v_0, v_1) \leftarrow e_2], \sigma^{\mathbb{SV}}[(v_0, v_1) \leftarrow v_2]) \in \gamma_\Sigma(\mathbf{assign}^\sharp_{\mathbb{E}}(e_0, e_1, e_2, \sigma^\sharp))$$

To achieve this, both the type and the formula properties of zones may need to be updated: (1) information about cells that may be overwritten should be removed from the abstract state, either by removing existing zone predicates, or by splitting zones, and removing the assigned cell from them; (2) type and formula information should be synthesized and attached to the zone corresponding to the cell that is modified in the assignment. Especially, type information is obtained by evaluating the semantics of abstract formulas; when this evaluation fails, a typing error should be reported.

Example 6 (Abstract assignment). Let us consider abstract state $\sigma^\sharp = (\mathtt{i} < \mathtt{n}, \{([1, \mathtt{i}-1] \times 2, e^\sharp), ([\mathtt{i}, \mathtt{n}] \times 2, \text{“} = \mathbf{String}\text{”})\})$ (type information is omitted for the sake of simplicity), where $e^\sharp = \mathbf{Int} + \mathbf{C}[+0, -1]$. Then, assignment $\mathbf{C}[\mathtt{i}, 2] =$ “$= 24 + \mathbf{C}[\mathtt{i}, 1]$” replaces the constant formula (of type string) contained in cell $\mathbf{C}[\mathtt{i}, 2]$ with a formula that can be abstracted by $\mathbf{Int} + \mathbf{C}[\mathtt{i}, 1]$ (or equivalently $\mathbf{Int} + \mathbf{C}[+0, -1]$), and it evaluates that formula, which returns a value of type **Int**. Thus, the string constant value that was previously stored in the cell is replaced, so the topmost cell of zone $[\mathtt{i}, \mathtt{n}] \times 2$ should be removed from that zone. Therefore, we obtain abstract state $\sigma^\sharp_0 = (\mathtt{i} < \mathtt{n}, \{([1, \mathtt{i}-1] \times 2, e^\sharp), (\mathtt{i} \times 2, e^\sharp), ([\mathtt{i}+1, \mathtt{n}] \times 2, \text{“} = \mathbf{String}\text{”}), (\mathtt{i} \times 2, \mathbf{Int})\})$.

Condition Test, Reduction, Join and Widening. The use of these operators in the construction of abstract semantics are classic. We summarize their signature and soundness condition as follows:

$\mathbf{guard}^\sharp : \mathbb{E} \times \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$	$\forall \sigma \in \gamma_\Sigma(\sigma^\sharp),\ \mathbf{true} \in [\![e]\!]_{\mathbb{E}}(\sigma) \Rightarrow \sigma \in \gamma_\Sigma(\mathbf{guard}^\sharp(e, \sigma^\sharp))$
$\mathbf{reduce}^\sharp : \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$	$\forall \sigma^\sharp \in \mathbb{D}^\sharp_\Sigma,\ \gamma_\Sigma(\sigma^\sharp) \subseteq \gamma_\Sigma(\mathbf{reduce}^\sharp(\sigma^\sharp))$
$\sqcup^\sharp : \mathbb{D}^\sharp_\Sigma \times \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$	$\forall \sigma^\sharp_0, \sigma^\sharp_1 \in \mathbb{D}^\sharp_\Sigma,\ \gamma_\Sigma(\sigma^\sharp_0) \cup \gamma_\Sigma(\sigma^\sharp_1) \subseteq \gamma_\Sigma(\sigma^\sharp_0 \sqcup^\sharp \sigma^\sharp_1)$
$\nabla^\sharp : \mathbb{D}^\sharp_\Sigma \times \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$	$\forall \sigma^\sharp_0, \sigma^\sharp_1 \in \mathbb{D}^\sharp_\Sigma,\ \gamma_\Sigma(\sigma^\sharp_0) \cup \gamma_\Sigma(\sigma^\sharp_1) \subseteq \gamma_\Sigma(\sigma^\sharp_0 \nabla^\sharp \sigma^\sharp_1)$

The reduction [11] operator identifies zones that can be merged by generalizing bounds using numerical relations. Additionally, as absolute cell references can be rewritten into relative cell references, and vice versa, with the help of zone information, similar abstract formulas may be identified to permit a reduction.

Example 7 (Reduction). (1) In abstract state $\sigma^\sharp_0$ of Example 6, $([1, \mathtt{i}-1] \times 2, e^\sharp)$ and $(\mathtt{i} \times 2, e^\sharp)$ can be merged into $([1, \mathtt{i}] \times 2, e^\sharp)$. (2) As $(4 \times 4, \mathbf{C}[4, 3] * \mathbf{Float})$ can be rewritten to $(4 \times 4, \mathbf{C}[+0, -1] * \mathbf{Float})$, and $(5 \times 4, \mathbf{C}[5, 3] * \mathbf{Float})$ can be

rewritten to $(5 \times 4, \mathbf{C}[+0, -1] * \mathbf{Float})$, these two zones can be merged into one zone $([4, 5] \times 4, \mathbf{C}[+0, -1] * \mathbf{Float})$.

The abstract join $\sqcup^\sharp$ needs to identify zones of both arguments that are paired with the same cell abstract property, and then to attempt generalizing the zone bounds up to the point where an intersection can be found that does not lose too much precision (indeed, $(\mathcal{Z}_0, \mathcal{P})$ and $(\mathcal{Z}_1, \mathcal{P})$ can be over-approximated by $(\mathcal{Z}, \mathcal{P})$ where $\gamma_{\text{num}}(\mathcal{Z}) \subseteq \gamma_{\text{num}}(\mathcal{Z}_0)$ and $\gamma_{\text{num}}(\mathcal{Z}) \subseteq \gamma_{\text{num}}(\mathcal{Z}_1)$). The definition of the widening [10] operator $\nabla^\sharp$ is similar, except that abstract predicates of unstable zones are removed completely after a fixed rank to guarantee termination of all sequences of abstract iterates. The formula zones shown in Fig. 2 are derived thanks to this widening operator.

Example 8 (Abstract join). Let us consider abstract states $\sigma_0^\sharp = (\mathtt{x} = 2, \{(\mathcal{Z}_0, e^\sharp)\})$ and $\sigma_1^\sharp = (\mathtt{x} = 3, \{(\mathcal{Z}_1, e^\sharp)\})$, where $\mathcal{Z}_0 = [1, 2] \times 2$ and $\mathcal{Z}_1 = [1, 3] \times 2$. In both zones, the upper bound on i is equal to $\mathtt{x}$. Thus, $\mathcal{Z}_0$ (resp., $\mathcal{Z}_1$) is semantically equivalent to $\mathcal{Z} = [1, \mathtt{x}] \times 2$. Therefore, $\sigma_1^\sharp \sqcup^\sharp \sigma_0^\sharp$ returns $(2 \leq \mathtt{x} \leq 3, \{(\mathcal{Z}, e^\sharp)\}$.

5.2 Static Analysis of Common Program Statements

We now write $\mathbf{lfp}^\sharp$ for the classic operator which computes an abstract post-fixpoint for any given abstract function, using widening operator $\nabla^\sharp$. If $F : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, $F^\sharp : \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$, $\mathcal{S} \subseteq \Sigma$ and $\sigma^\sharp \in \mathbb{D}^\sharp_\Sigma$ are such that F is continuous, $F \circ \gamma_\Sigma \subseteq \gamma_\Sigma \circ F^\sharp$ and $\mathcal{S} \subseteq \gamma_\Sigma(\sigma^\sharp)$, then $\mathbf{lfp}_\mathcal{S} F \subseteq \gamma_\Sigma(\mathbf{lfp}^\sharp_{\sigma^\sharp} F^\sharp)$.

We can now describe the abstract semantics of all the statements in the following table, except for global re-evaluation instruction **Eval**, which will be described in Sect. 5.3:

$$[\![s_0;\ s_1]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = [\![s_1]\!]^\sharp_\mathbb{P}([\![s_0]\!]^\sharp_\mathbb{P}(\sigma^\sharp)) \quad [\![\mathtt{x} = e]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = \mathbf{reduce}^\sharp(\mathbf{assign}^\sharp_\mathbb{X}(\mathtt{x}, e, \sigma^\sharp))$$
$$[\![\mathbf{C}[e_0, e_1] = e_2]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = \mathbf{reduce}^\sharp(\mathbf{assign}^\sharp_\mathbb{V}(e_0, e_1, e_2, \sigma^\sharp))$$
$$[\![\mathbf{C}[e_0, e_1] = \text{“} = e_2\text{”}]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = \mathbf{reduce}^\sharp(\mathbf{assign}^\sharp_\mathbb{E}(e_0, e_1, e_2, \sigma^\sharp))$$
$$[\![\mathbf{If}\ e\ \mathbf{Then}\ s_0\ \mathbf{Else}\ s_1\ \mathbf{End}]\!]^\sharp_\mathbb{P}(\sigma^\sharp)$$
$$= [\![s_0]\!]^\sharp_\mathbb{P}(\mathbf{reduce}^\sharp(\mathbf{guard}^\sharp(e, \sigma^\sharp))) \sqcup^\sharp [\![s_1]\!]^\sharp_\mathbb{P}(\mathbf{reduce}^\sharp(\mathbf{guard}^\sharp(\neg e, \sigma^\sharp)))$$
$$[\![\mathbf{While}(e)\ s\ \mathbf{End}]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = \mathbf{reduce}^\sharp(\mathbf{guard}^\sharp(\neg e, \mathbf{lfp}^\sharp_\bot F^\sharp))$$
$$\text{where } F^\sharp(\sigma_0^\sharp) = \sigma^\sharp \sqcup^\sharp [\![s]\!]^\sharp_\mathbb{P}(\mathbf{reduce}^\sharp(\mathbf{guard}^\sharp(e, \sigma_0^\sharp)))$$

Moreover, as the abstract semantics of each statement is sound, the global soundness theorem can be proved by induction over the syntax of programs:

Theorem 2 (Soundness). *For all statements $s \in \mathbb{P}$, $[\![s]\!]^\sharp_\mathbb{P}$ is sound: $\forall \sigma^\sharp \in \mathbb{D}^\sharp_\Sigma$, $\forall \sigma \in \gamma_\Sigma(\sigma^\sharp)$, $[\![s]\!]_\mathbb{P}(\sigma) \subseteq \gamma_\Sigma([\![s]\!]^\sharp_\mathbb{P}(\sigma^\sharp))$. Thus, for all $a \in \mathbb{A}$, $[\![a]\!]^\sharp_\mathbb{A}$ is sound, i.e., $[\![a]\!]_\mathbb{A} \subseteq \bigcup \gamma_\Sigma([\![a]\!]^\sharp_\mathbb{A})$.*

Therefore, the whole analysis catches all typing errors following the definition given in Sect. 3, corresponding to the operations specified unsafe. In particular, it catches the error of the example shown in Sect. 2, and proves the fixed version safe.

5.3 Static Analysis of Global Re-evaluation

The concrete semantics of **Eval** involves a global re-computation of the spreadsheet cell values while preserving formulas and variables values (Sect. 3). Therefore, we assume $\sigma^\sharp = (N^\sharp, F^\sharp \uplus T^\sharp)$, and show the computation of $T^\sharp_{\text{res}}$ so as to let $[\![\mathbf{Eval}]\!]^\sharp(\sigma^\sharp) = (N^\sharp, F^\sharp \uplus T^\sharp_{\text{res}})$. This computation proceeds by fixpoint approximation. We show several abstract iteration strategies.

Cell-by-cell Re-evaluation. In the following, we aim at setting up an abstract counterpart of $\mathcal{F}_\prec$, given a cell order $\prec$ compatible with formula dependencies. In order to show a simple case of abstract iteration, we first make the assumption that all abstract zones are reduced to exactly one concrete cell, thus elements of $T^\sharp$ (resp., $F^\sharp$) are equivalent to functions from $\mathbb{R} \times \mathbb{C}$ into $\mathbb{T}$ (resp., $\mathbb{E}^\sharp$). We also remark that abstract formulas follow the same dependencies as concrete formulas; the topological order $\prec$ can be retrieved from an abstract state.

Then, we need to extend our existing functions with $\bot$, which means a cell is not yet evaluated:

- We extend type spreadsheets $\mathbb{S}_\mathbb{T}$ and let $\mathbb{S}_{\mathbb{T}\bot} = (\mathbb{R} \times \mathbb{C}) \to (\mathbb{T} \uplus \{\bot\})$, and let order relation $\sqsubseteq$ be defined by $\forall t \in \mathbb{T},\ \bot \sqsubseteq t$; As each zone contains exactly one cell, an element $T^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,type}})$ is now equivalent to an element of $\mathbb{S}_{\mathbb{T}\bot}$.
- We let $T^\sharp_\bot \in \mathbb{S}_{\mathbb{T}\bot}$ be defined by $\forall (i,j),\ T^\sharp_\bot(i,j) = \bot$;
- Given an abstract formula $e^\sharp$, the abstract semantics $[\![e^\sharp]\!]_{\text{t}}$ can also be extended to compute a type (possibly $\top$) for an abstract element of $\mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,form}})$, using the type information available for each cell in the formula; we still use notation $[\![e^\sharp]\!]_{\text{t}}$ to denote that extended semantics.

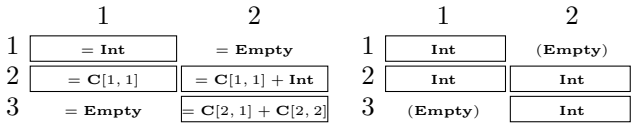
Now we have all the elements to define $\mathcal{F}^\sharp_\prec : \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,type}}) \to \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,type}})$ as the abstract counterpart of $\mathcal{F}_\prec$. It is such that for all $T^\sharp_0 \in \mathbb{S}_{\mathbb{T}\bot}$, and for all i, j, $\mathcal{F}^\sharp_\prec(T^\sharp_0)(i,j) = t_1$, where $t_1 = [\![F^\sharp(i,j)]\!]_{\text{t}}(T^\sharp_0)$ when $T^\sharp_0(i,j) = \bot$ and $\forall (i',j') \prec (i,j),\ T^\sharp_0(i',j') \neq \bot$; otherwise $t_1 = T^\sharp_0(i,j)$. Then, we prove: $\forall \sigma^\sharp_0 = (N^\sharp, F^\sharp \uplus T^\sharp_0) \in \mathbb{D}^\sharp_\Sigma,\ \forall (\sigma^\mathbb{X}, \sigma^\mathbb{SE}, \sigma^\mathbb{SV}_0) \in \gamma_\Sigma(\sigma^\sharp_0), (\sigma^\mathbb{X}, \sigma^\mathbb{SE}, \mathcal{F}_\prec(\sigma^\mathbb{SV}_0)) \subseteq \gamma_\Sigma(N^\sharp, F^\sharp \uplus \mathcal{F}^\sharp_\prec(T^\sharp_0))$.

Therefore, the existence of the fixpoint follows from the continuity of $\mathcal{F}^\sharp_\prec$ (it is obtained after at most $n_\mathbb{R} \cdot n_\mathbb{C}$ iterations). Soundness is proved by fixpoint transfer:

Theorem 3 (Abstract interpretation of re-evaluation). $[\![\mathbf{Eval}]\!]^\sharp_\mathbb{P}(N^\sharp, F^\sharp \uplus T^\sharp) = (N^\sharp, F^\sharp \uplus T^\sharp_{\text{res}})$ *where* $T^\sharp_{\text{res}} = \mathbf{lfp}^\sharp_{T^\sharp_\bot} \mathcal{F}^\sharp_\prec = \bigsqcup\{(\mathcal{F}^\sharp_\prec)^n(T^\sharp_\bot) \mid n \in \mathbb{N}\}$ *satisfying the soundness condition:* $\forall \sigma^\sharp \in \mathbb{D}^\sharp_\Sigma,\ \forall \sigma \in \gamma_\Sigma(\sigma^\sharp),\ [\![\mathbf{Eval}]\!]_\mathbb{P}(\sigma) \subseteq \gamma_\Sigma([\![\mathbf{Eval}]\!]^\sharp_\mathbb{P}(\sigma^\sharp)$

Moreover, this process will also allow us to prove no typing error (in the sense of Sect. 3) arises during re-evaluation.

Example 9 (Cell-by-cell re-evaluation). We illustrate this strategy with the abstraction of the spreadsheet studied in Sect. 3. The corresponding abstract formulas over zones are shown below, in the left hand side. Then, cells are treated following topological ordering $(1,1) \prec (2,1) \prec (2,2) \prec (3,2)$, and the type obtained for each cell is **Int**:

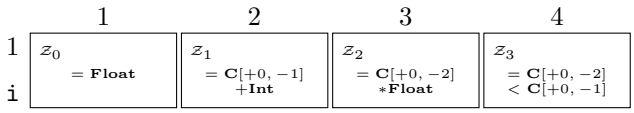


Zone-by-zone Strategy. While the cell-by-cell strategy illustrates abstract interpretation of **Eval** well, it is not effective in practice, as abstract states usually contain zones which are bounded, but possibly large and/or of variable size. Thanks to the abstraction of a zone where abstract formulas are unique, very often type information can be directly computed over a whole zone, which is much faster than evaluation of each of their cells.

The evaluation of a zone can be undertaken as soon as 2 conditions are satisfied: (1) its abstract formulas induce no dependency inside it. In the example of Sect. 2, this holds true for all zones, but the last column (which will be discussed in the next paragraph); (2) there exists a topological order $\prec$ compatible with formula dependencies, according to which all the cells lower than the cells in that zone have already been evaluated.

Once a zone is identified to be evaluated, we can follow its unique abstract formula, apply the functions to the types of its arguments (which have been evaluated), and return the new types for this zone. (For both the arguments and the result, their type is not necessarily unique. e.g., it is possible that the result consists of several sub-zones with different types.) As the evaluation of zone is efficient, we always try to find available zones for evaluation, which boils down to the computation of zones one by one.

Example 10 (Zone-by-zone strategy). We assume the following abstract formula information is available:



Then according to the formula dependencies, the abstract iteration can follow an order of $\mathcal{Z}_0, \mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3$. It terminates after four iterations, and produces the type zones $\{(\mathcal{Z}_0, \mathbf{Float}), (\mathcal{Z}_1, \mathbf{Float}), (\mathcal{Z}_2, \mathbf{Float}), (\mathcal{Z}_3, \mathbf{Bool})\}$.

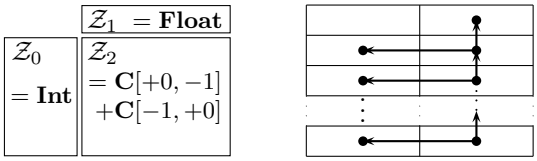
Abstract Iteration over Zones, Using Widening. Not all cases can be handled by the zone-by-zone strategy. One case is where abstract formulas induce dependency inside a zone. For instance, in the example of Sect. 2, the evaluation of Column 5 requires types of Columns 3, 4 and Column 5 itself, we call this *self-reference*. *Inter-reference* among zones is also possible, where $\mathcal{Z}_0$ needs types of $\mathcal{Z}_1$, $\mathcal{Z}_1$ needs types of $\mathcal{Z}_2$, ..., and $\mathcal{Z}_n$ needs types of $\mathcal{Z}_0$.

Although self-reference or inter-reference may exist among zones, there is still no circular dependency among cells. This suggests to split complex zones that contain dependencies by pulling off the first cell in the dependencies order and treating it first (though this cell may not be unique). This process can then be iterated to treat all cells in the zones. The strategy follows the steps below:

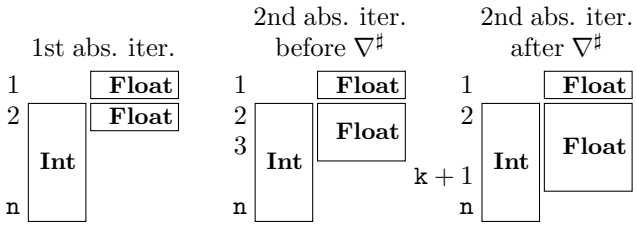
1. introduce variable `k`, denoting the number of cells in the abstract formula zone that have been re-evaluated;
2. determine the first cell in the order of dependencies in the abstract formula zone;
3. split the abstract formula zones into two parts: the above cell that can be immediately evaluated, and the rest of the abstract formula zone to be analyzed;
4. iterate Steps 2 and 3 until the abstract formula zones are fully treated, and apply widening at each step to ensure termination (thus, the analysis of **Eval** utilizes on $\mathbf{lfp}^\sharp$);
5. when Step 4 produces stable type zones, synthesize the final abstract state by restricting and integrating the stable type zones.

This strategy provides a way to compute the effect of **Eval** over large zones or zones of variable size. It does not need the full unrolling of the zone, thanks to the use of the widening operation over the cells that are generally well structured. Indeed, it effectively amounts to analyzing a loop with counter `k` that iterates over the zone in order to compute abstract types.

Example 11 (Abstract iteration over a zone). We consider the abstract formula zones below (which correspond to an excerpt of the example of Sect. 2), which define the dependencies shown in the right-hand side:



The first two iterations of the strategy described above produce the results below:



A third iteration proves this abstract state stable; furthermore, it establishes all formulas in the zone evaluate without a typing error and produce a result of type **Float**.

Combined Strategies. In general, abstract states require the use of a combination of the strategies shown above. The zone-by-zone strategy is given priority in our analysis: we try to detect, as much as possible, zones which can be evaluated as a whole. For most of spreadsheet applications in practice, just following zone-by-zone strategy is sufficient to evaluate all the cells; in case, widening and cell-by-cell strategies can complete. Therefore, their composition achieves global re-evaluation, which is sound. After adding **Eval** statement to the set of the statements, the global soundness theorem (Sect. 5.2) still holds.

6 Implementation of an Excel VBA Analyzer

We have implemented our analysis. Our analyzer handles a frequently used subset of Microsoft Excel functions and VBA, following the VBA specification [2]. Given an Excel file, the first part of our tool (written in VBA) opens it within Microsoft Excel and parses the contents of the spreadsheet tables (e.g., number formats, types, formulas, buttons) and VBA macros; the second part of our tool (19000 lines of OCaml code) undertakes the verification. The verification of a spreadsheet application proceeds through two steps: (1) the verification of global re-evaluation; (2) the verification of the execution of any macro it contains, given the initial spreadsheet abstract state. The verification gradually infers invariants; finally, it either proves the correctness with regard to our typing system, or raises alarms by pointing out the location (e.g., the zone in spreadsheets and/or the line in macros) and the unsafe typing rule in question. The analyzer can also be launched over a set of Excel files and return a summary report for the whole set. We will present the analysis results for the EUSES Spreadsheet Corpus in Sect. 7.

Supported Features. In the previous sections, we focused on the core spreadsheet language to formalize the abstraction and the analysis. Actually, in order to analyze real-world spreadsheets, our implementation supports many more spreadsheet features, including the following:

- A *workbook* may contain several worksheets, and a *formula* may refer to cells in another sheet or another workbook.
- Macros may contain *interprocedural calls*: another user-defined subroutine or function can be called with or without arguments.
- *Number formats* are options that Excel provides for displaying values such as percentages, currencies, dates, which impact value types in some cases. Therefore, we also abstract this information (using zones as well) and take it into account while typing.

Circular References. The spreadsheet environment we have formalized does not feature circular references among cells, yet Microsoft Excel allows circular references under certain circumstances. In particular, a number of iterations can be set so that a circular computation could terminate. In this case, both the starting cell and the ending cell of the evaluation can be identified. Following this order, the analyzer iterates the abstract evaluation until it reaches a fixpoint.

Data Validation. Excel users may define constraints on data to be entered in some areas, such as "empty or only date", "empty or only time", "only text of a certain length", etc. Such information constrains data to be written in some areas at run-time; thus, this information can be used in the analysis. Therefore, our analyzer parses areas with data validation constraints and uses the type information they provide to the initial abstract state of the spreadsheet. This allows a precise verification of spreadsheets that utilize data unknown at verification time / nondeterministic data.

Over-approximation of Empty Input Cells. Spreadsheet formulas may refer to empty cells where values will be entered by users later. If "Data Validation" is not available for these cells, we can still derive their "expected" type from the function that is applied to them. For instance, function `SUM` expects **Numeric** arguments, function `AND` expects **Bool** arguments, etc. To account for this, the analyzer will either treat these cells as empty or store a value of that type. This over-approximation helps better verify formulas / macros using those cells.

7 Experiments and Analysis Results

We evaluated the efficiency of our tool, and focused on the three following questions:

1. Does the analysis find real defects in spreadsheets & macros?
2. How long does the analysis take?
3. Is the analysis report precise enough? Is it easy enough for users to diagnose and adopt?

Experimental Setup. We chose the EUSES Spreadsheet Corpus [15] as an experimental subject for two reasons. First, to the best of our knowledge, it is the largest publicly available sample of real-world spreadsheets. Secondly, it includes many macros that offer good candidates for evaluating our associated program analysis. The sizes of the files of the corpus range from several KB to dozens of MB. In general, the spreadsheets are no longer in the stage of development and are already operational.

A spreadsheet may contain zero, one or several macros. It may also not contain any formulas. The following table presents the classification of the EUSES Spreadsheet Corpus. Category D corresponds to pure data-sheets without any formulas or macros: they are not meaningful for our analysis, as their analysis is trivial. Therefore, our sample was the 2120 spreadsheets of Categories A + B + C and the 1053 macros inside them.

A	# spreadsheets with ≥ 1 formulas & 0 macro	1959
B	# spreadsheets with ≥ 1 formulas & ≥ 1 macros	111
C	# spreadsheets with 0 formula & ≥ 1 macros	50
D	# spreadsheets with 0 formula & 0 macro	2532

We performed the experiments as follows. First, our tool parsed all the spreadsheets and the macros, and detected 27 macros and 59 spreadsheet tables that have syntactic bugs or are incomplete (e.g., users put evident annotations such as "not-available" in their spreadsheet where an analysis would not be relevant). Next, we launched the analyzer on the rest of the items, and it was able to analyze **Eval** for 1854 spreadsheets and 858 macros (the reason why 7.8% of the spreadsheet tables and 16.4% of the macros were not analyzed is due to the fact our tool currently does not handle all Excel & VBA features and built-in functions, which are quite complex and numerous). Last, we filtered out the items whose bugs are not type-related (e.g., calls to undefined macros). Our tool detected 15 such spreadsheets and 21

such macros, which is useful but orthogonal to our purpose. The reste of the analyzed items are either type-related safe or erroneous, we classify them by Category **TypeRSE** in the following table, which summaries the analyses of **Eval** of spreadsheets in Categories A + B and macros in spreadsheets of Categories B + C. From now on we shall focus on Category **TypeRSE** and discuss the core of the analysis.

	Total			
	Syntactically Erroneous or Incomplete	Syntactically Correct		
		Non-Analyzed	Analyzed	
			Type-Unrelated Erroneous	**TypeRSE**
Eval	59	157	15	1839
Macro	27	168	21	837

Real Defects. We had set up the analyzer in such a way that, once an unsafe typing rule is applied, it raises an alarm which stops the verification. In this case, when we ran the analyzer on the EUSES Corpus, the number of alarms raised was also the number of spreadsheets / macros that were considered potentially erroneous by our analysis. In total, the analyzer raised 69 type-related alarms for **Eval** and 73 for macros. For each alarm, the report specified its location (e.g., the zone in spreadsheets and/or the line in macros), the unsafe typing rule (bug pattern) it encountered, and an estimate rating of how severe the defect would be. We manually inspected the spreadsheet / macro for which an alarm was raised, to diagnose its cause and the consequence of the revealed problem.

The alarms of type-unsafe operations effectively led us to identify *real defects* in programs, part of which defects silently produce wrong results. We show some of them as examples:

Example 1. In "homework\processed\Finalgradebook.xls", an application of Function `AVERAGE` to an Empty zone was detected, whereas all of its other arguments were Double. We noticed that a formula had been coded as "=`AVERAGE`(D4;F4;H4;J4;L40)", where "L40" referred to an empty cell. This was probably due to a user's erroneous typing of "L40" instead of "L4" (which was a Double and should have been an argument of `AVERAGE`), whereas Excel permitted the formula by discarding the empty cell. This mistake will indeed result in the computation of an incorrect average value.

Example 2. In "modeling\processed\2-26.xls", subroutine "do_assign" uses a two-level loop to copy a table of basic parameters into another sheet where biological simulations are performed. Our tool detected that the whole zone of the table was Double except the first line, which was Empty. However, this zone had been assigned to a Double zone in another sheet. Upon investigation, we noticed a one-line shift between the source table and the target table, because the range of the loop was wrong. This will result in the target table being incorrectly filled in (its first line filled in with 0s, the copy result of empty cells), and the simulations (run 100 times!) based on these parameters will generate incorrect results.

Example 3. In "homework\processed\pl_student2002.xls", the analyzer detected an application of Function `SUM` to a String value, whereas all of its other arguments were Double. Examining the spreadsheet, we observed that the String

value was actually "I", whereas the other arguments were either 0 or 1. Clearly users had mistaken "I" and "1", which are visibly similar. As a result, Excel considers "I" as 0 by `SUM`, which leads to a different number from that originally intended.

As shown by these examples, our tool discovered defects that would be hard for users to spot. In total, among all the alarms raised by the analyzer, we identified 25 real defects for **Eval** and 20 for macros, corresponding to serious and harmful issues in spreadsheet applications.

Among patterns contributing to spreadsheet defects, we can cite: (1) binary operation on Numeric data and Non-Numeric data (e.g., String) (2) Non-Numeric data (e.g., String, Empty) among the arguments of `SUM` or `AVERAGE`.

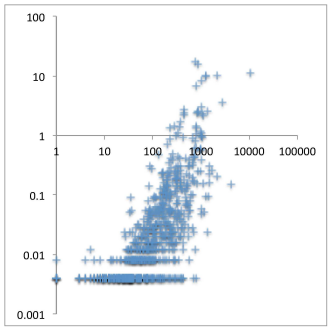
Furthermore, the defects found in the programs can be classified into several major categories: (1) Formulas or statements are applied to a wrong sheet area, and consequently take unexpected arguments. In Example 1, it is the reference of an argument of the formula that is incorrect; in Example 2, the area of the copied table is wrongly set. This kind of defect typically occurs due to an inappropriate manipulation (e.g., mistyping, improper copy-paste). In total, we found 13 bugs of this category. (2) Formulas or statements have a certain assumption for the types or the values of input data, yet the assumption is not specified or will not always hold at run-time. For instance, in a macro of "homework\processed\RT_EvaluationWorkbook.xls", an addition of a String value and an Empty cell is involved, and the analyzer realized that the Empty cell (representing reference of products) could well be set to a number at run-time, which would block the execution of the macro. We detected 8 defects of this class.

Moreover, we observe that many real defects were found thanks to the abstraction of the initial state of the spreadsheets, since this abstraction takes into account data that will be entered at run-time (Data-Validation areas, functions reading external values, etc.). It is, for instance, the case of the error in "RT_EvaluationWorkbook.xls", where the over-approximation of an empty cell covers numeric data at run-time, which is not the current value of the given spreadsheet. This kind of error would not be discovered by verification techniques that rely on a single spreadsheet state, like testing.

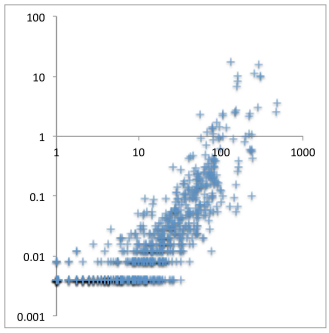
Analysis Time. The analyzer succeeded in verifying 858 macros in 161 spreadsheets (Categories B + C in Table 7). The size of each macro ranges from a few LOCs to several hundred LOCs. As one LOC could well involve a complex abstract operation by executing a complex statement or calling another macro, the size of a macro is just one of the factors that have an impact on its analysis time. We can list other important factors such as the complexity of the abstract state (e.g., # formula zones, # type zones, # variables) and the number of complex abstract operations (e.g., join, widening, reduction, eval). By summarizing all of the 858 successfully analyzed macros, we observe that the analysis for macros is fast enough: only 2% of them lasted more than 3 seconds (the longest analysis takes 10.45 seconds), and 88% of them took less than 0.2 second. We note that all analyses with fewer than 100 abstract zones and no loop of nesting depth greater than 2 lasted less than 1.75 seconds.

Figure 4(a) indicates the analysis time for **Eval** against the number of cells for non-constant values in initial spreadsheets. We remark that the analyses were performed in a reasonable time frame: 99% of the analyses took less than 1 second. Thus, the analysis time is acceptable in practice, and the analysis would integrate in a seamless manner in development.

Additionally, Figure 4(b) shows the analysis time of **Eval**, against the number of abstract formula zones for non-constant values in initial spreadsheets. By comparing it with Fig. 4(a), we remark that the principal attribute for analysis time is the number of abstract zones, rather than the number of cells. This observation is consistent with our abstraction mechanism, which is based on a cardinal power of zone abstractions. Going further, we remark that, on average, the number of zones we have made is 0.1x as many as the number of cells for a spreadsheet. The larger a spreadsheet is, the lower this ratio is: for certain large spreadsheets, this ratio can be less than 0.01. This guarantees that our analysis based on zones is scalable and especially efficient for large spreadsheets.



(a) vs. # cells for non-constant values



(b) vs. # abstract formula zones for non-constant values

Fig. 4. Analysis time for **Eval** in seconds

Precision and Diagnostics. With regard to our current typing system, the analyzer proved that global re-evaluations of 1770 spreadsheet tables in Categories A + B of Table 7 were correct and that 764 macros were correct.

When it raises an alarm, the analyzer issues a report including the context information (zone, macro line) and the category of the potential defect. In addition, Excel & Visual Studio provide an interactive debugging environment where the states of spreadsheets and program variables are highly visible. Thus, users can assess the alarm reports interactively with the help of this environment.

Besides the categories of real defects we presented previously, we can list several major categories of false alarms: (1) The first category is due to imprecisions in the analysis: the over-approximation causes the alarms corresponding to unsafe

concrete states that will never be reached. We notice that the majority of the false alarms in this category come from the imprecisions in the analysis of certain VBA and Excel built-in functions. Few of the false alarms for **Eval** are due to the over-approximation of the initial state of the spreadsheet. This implies that a technique that relies on the given state of the spreadsheet would not reduce these false alarms. (2) The second category of false alarms is indeed related to type-unsafe operations, that are intended as such by the users. For example, sometimes users apply Function `SUM` to a column containing not only data, but also several titles. In this case, Function `SUM` will omit the titles and will thus still produce the correct result, summing the numeric data only. Yet, this pattern will result in false alarms due to Non-Numeric data among the arguments of `SUM`.

Therefore, diagnosing an alarm and triaging it as a false alarm or a real defect is fairly straightforward and typically takes a couple of minutes. We spent no more than 10 minutes on the most complex alarms. In total, we identified 44 false alarms for **Eval** and 53 for macros. The following table summarizes the core analyses.

	TypeRSE (Type-Related Safe or Erroneous)		
	alarm free	raise alarm	
		real defect	false alarm
Eval (1839)	1770	25	44
Macro (837)	764	20	53

Overall, the tool raised 142 alarms from 2676 analyses (**Eval** + macros), 45 of which alarms (i.e., approximately 30%) were identified as real defects, which makes the false alarm number quite acceptable, considering that the defects found would be hard to spot by simple testing.

Summary. The experiments on the EUSES Corpus show that our analysis succeeds in detecting type-unsafe operations and can effectively be used to improve the quality of spreadsheets. It discovers defects that will cause unexpected results and that will not likely be found by testing. The diagnosis of alarms is not a tedious process with the guidance of the tool, and the false alarm number is reasonable. While the zone abstractions of a spreadsheet allows for the verification of type properties, it makes the analysis scalable for spreadsheets having a large number of cells. The analysis is efficient enough to be integrated within a development environment, as it could either be scheduled as a background task (e.g., scan systematically before saving), using reasonable resources, or launched upon user request in an interactive way.

8 Related Work

Unit Verification. The existing projects [3, 9, 4–6] resolve concrete units or dimensions with labels, headers and/or other annotations, build typing systems and reason about the correctness of formulas. We cannot find their experimental data or precision reports on comparable sets of benchmarks for a practical comparison

with our results. Nevertheless, theoretically, our work is different from theirs in several ways: (1) the types we treat are classical from a programming language point of view, whereas their types refer to the concrete meaning of objects; thus, the built-in rules or bug patterns of the two analyses are not the same; (2) we verify both the interface level and associated programs that the existing projects do not consider; (3) we evaluate formulas according to their order of precedence and thus support spreadsheets where data may be outdated; (4) our system covers a larger library of spreadsheet functions; (5) our classical types can always be retrieved from spreadsheets. By contrast, given a spreadsheet, the concrete meaning of objects are not always clear, and the retrieval of these meanings relies on annotation, though the analysis can be finer-grained if the retrieval is successful (e.g., they detect "adding apples and oranges", which our analysis does not regard as an error). Actually, combining our work with that of the existing projects would be a good direction for future work. By substituting other lattices with the type lattice and merging typing systems, we would be able to perform finer-grained analyses with various units and types.

Array Analysis and Zone Domain. Array analyses such as [23, 12] also tie abstract properties to array regions; a notion of dependent types has been used to specify array properties such as array size [27]. One difference of our work is that we treat bi-dimensional arrays, whereas the existing work studies uni-dimensional arrays.

Cheng and Rival [8] introduce an abstract domain to describe zones in two-dimensional arrays and apply it to analyze programs in a limited language. We aim at verifying real-world spreadsheets, which consist of associated programs and formulas.To this end, we formalize a larger spreadsheet language which embraces formulas, and propose a global abstraction by tying abstract formulas and types to zones. The execution of the formulas is the global re-evaluation, which can be launched in an associated program, or separately upon user request. We study several strategies to perform its analysis. Last, we evaluate the analysis and the implementation by analyzing a large set of real-world spreadsheets.

Eval. Jensen *et al.* [18] address the eval function in JavaScript, which dynamically constructs code from text strings and executes it as if it were regular code in ways that obstruct existing static analyses. The eval function in spreadsheets, which is an evaluation of the pre-coded spreadsheet formulas, has a very different semantics; thus, the approach in this paper cannot be directly related. Hammer *et al.* [16] propose a demand-driven incremental computation semantics of eval to provide speedups in spreadsheets, whereas our abstraction is based on the original concrete semantics of **Eval** in spreadsheets.

9 Conclusion

We have proposed a static analysis which is able to detect a significant class of subtle spreadsheet defects. It discovers inappropriate applications of operators and functions to arguments, which may produce unexpected results. To the best of our

knowledge, our analysis is the first that can handle spreadsheet formulas, global re-evaluation and associated programs. Our evaluation on the EUSES Corpus has demonstrated that our analysis can effectively run on real-world spreadsheet applications and can verify a large number of them. It is able to discover defects that would be beyond the reach of both testing techniques and static analyses that would ignore the dynamic aspects of spreadsheets.

References

1. Report of JPMorgan Chase & Co. management task force regarding 2012 CIO losses (Jan 2013)
2. [MS-VBAL]: VBA language specification. Tech. rep., Microsoft Corporation (Apr 2014)
3. Abraham, R., Erwig, M.: Header and unit inference for spreadsheets through spatial analyses. In: VL/HCC. IEEE Computer Society (2004)
4. Abraham, R., Erwig, M.: UCheck: A spreadsheet type checker for end users. J. Vis. Lang. Comput. (2007)
5. Ahmad, Y., Antoniu, T., Goldwater, S., Krishnamurthi, S.: A type system for statically detecting spreadsheet errors. In: ASE (2003)
6. Antoniu, T., Steckler, P.A., Krishnamurthi, S., Neuwirth, E., Felleisen, M.: Validating the unit correctness of spreadsheet programs. In: ICSE (2004)
7. Burnett, M., Atwood, J., Walpole Djang, R., Reichwein, J., Gottfried, H., Yang, S.: Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm (2001)
8. Cheng, T., Rival, X.: An abstract domain to infer types over zones in spreadsheets. In: SAS. Springer (2012)
9. Coblenz, M.J., Ko, A.J., Myers, B.A.: Using objects of measurement to detect spreadsheet errors. In: VL/HCC (2005)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. ACM (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. ACM (1979)
12. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL. ACM (2011)
13. Cunha, J., Fernandes, J.P., Mendes, J., Hugo Pacheco, J.S.: Towards a catalog of spreadsheet smells. In: ICCSA. LNCS (2012)
14. Cunha, J., Saraiva, J., Visser, J.: Model-based programming environments for spreadsheets. Science of Computer Programming (2014)
15. Fisher II, M., Rothermel, G.: The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In: WEUSE (2005)
16. Hammer, M.A., Phang, K.Y., Hicks, M., Foster, J.S.: Adapton: Composable, demand-driven incremental computation. In: PLDI. ACM (2014)
17. Hermans, F., Pinzger, M., Deursen, A.v.: Detecting and visualizing inter-worksheet smells in spreadsheets. In: ICSE (2012)
18. Jensen, S.H., Jonsson, P.A., Møller, A.: Remedying the eval that men do. In: ISSTA. ACM (2012)

19. Jones, S.P., Blackwell, A., Burnett, M.: A user-centred approach to functions in Excel. In: ICFP. ACM (2003)
20. Miné, A.: The octagon abstract domain. HOSC (2006)
21. Panko, R.R.: What we know about spreadsheet errors. Journal of End User Computing (JEUC) (1998)
22. Rajalingham, K., Chadwick, D.R., Knight, B.: Classification of spreadsheet errors. In: EuSpRIG Symposium (2001)
23. Reps, T., Gopan, D., Sagiv, M.: A framework for numeric analysis of array operations. In: POPL. ACM (2005)
24. Sestoft, P.: Online partial evaluation of sheet-defined functions. EPTCS (2013)
25. Sestoft, P.: Spreadsheet Implementation Technology. Basics and Extensions. MIT Press (2014)
26. Wakeling, D.: Spreadsheet functional programming. JFP (2007)
27. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: PLDI. ACM (1998)