Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection*

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

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

Abstract. Spreadsheets are widely used, yet are error-prone. In particular, they use 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 engineering, statistics, finance and management. They combine a flexible tabular representation of data in two-dimensional tables mixing formulas and values with associated programs (or macros), written in specific languages. For instance, Microsoft Excel includes a version of Visual Basic for Applications (VBA), whereas Google Spreadsheets have Google Apps Script and LibreOffice Calc has LibreOffice Basic.

Unfortunately, spreadsheet applications are subject to numerous defects, and often produce incorrect results that do not match user understanding as shown in [21,22]. In 2013, the Task Force Report [1] quoted losses of billions of dollars due to errors in spreadsheet applications used in JPMorgan'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, therefore they now attract increasing attention from users, IT professionals, and from the research community. Approaches proposed so far include new languages [7] and

* The research leading to these results has received funding from the European Research Council under the European Union's seventh framework programme (FP7/2007-2013), grant agreement 278673, Project MemCAD.

J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 26–52, 2015.
DOI: 10.1007/978-3-662-46669-8_2

enhancements to functional features of spreadsheets with better language design [19,27] 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 [3,4,5,6,9] 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. Most existing works focus on the spreadsheet tables and ignore the associated programs despite them being a very important component of spreadsheet applications, which can have a significant impact on spreadsheet contents, either through function calls from a spreadsheet formula or through an execution of a subroutine launched by users. They are also massively present in industrial spreadsheet applications.

Verification Objective. Spreadsheet languages supply basic operators and functions to perform operations on values such as text, number, boolean, date and time to use in formulas and programs. The type system of spreadsheet languages is weak and rarely considers a type mismatch an error, even though that means unexpected or incorrect results may be produced instead. For instance, Microsoft Excel implicitly 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 "" $< n$ to **false**, yet the empty string does not have an obvious numeric value. More generally, type mismatches are common and rarely block the execution with an explicit error message such as #VALUE!. Thus, users develop and run spreadsheet applications in the environment where program defects can be hidden. Verifying a spreadsheet application is exempt of any such defect requires a *strong type discipline*, and *precise typing information* about formula operands should be inferred.

Static Analysis of Spreadsheets with Macros. Existing works focus on the spreadsheet contents, assume the data in the sheet are fully specified, and do not consider spreadsheet instances with different input data. Yet, industrial spreadsheet applications often handle non-deterministic or non-statically known input in the following cases: (1) input data may be left blank when the application is developed and entered at a later stage (Excel features "Data Validation" for such cases, which allows to specify restrictions on data before they are entered); (2) data may be defined dynamically, e.g., using functions generating random values, inserting values found on the Web or in external databases; (3) formulas and data edited in non-automatic calculation mode (i.e., when the spreadsheet environment does not always recalculate cell depending on the modified zone) may result in outdated values; (4) data and formulas may be set and manipulated by associated programs.

Therefore, in this paper, we propose a complete vision of spreadsheet applications, that includes spreadsheets storing *formulas* and *associated programs* (macros); they receive input data that is *unknown at verification time* (i.e., non-deterministic or read at run-time); their execution consists in *globally evaluating spreadsheet formulas* or *running an associated program*.

We propose a fully automatic and sound analysis, that discovers all type defects in spreadsheet applications. It features a strong type system, and an abstract domain that ties properties (like contents types) to *zones* in spreadsheet tables. It infers invariants by conservative abstract interpretation of spreadsheet applications. It either proves type correctness or displays potential issues to developers. Invariants also give a high-level view of program behaviors. 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 has the following benefits: (1) it unearths errors that dynamic tests may miss, as it computes an over-approximation of *all* the states executions can reach even in the presence of inputs at run-time; (2) it is efficient enough to be run during the development of a spreadsheet application. In this paper, we make the following contributions:

- We set up a concrete model for reasoning about spreadsheet applications, to be used as a basis for the definition and the proof of our analysis (Sect. 3);
- We propose an abstraction for spreadsheet applications, that takes the structure of formulas into account and is adapted to the type verification (Sect. 4);
- We define a static analysis, that takes into account both the contents of spreadsheet and the associated programs (Sect. 5), and is able to cope with global re-evaluation of spreadsheet contents (Sect. 5.2);
- We present our tool (Sect. 6) and report on results of verification of the EUSES Spreadsheet Corpus by our tool (Sect. 7).

2 Overview

In this section, we consider a realistic application, which silently produces wrong results that cannot be caught by the weak type system 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 table contains several columns storing asset variations and values expressed in two currencies, and computes the number of weekdays where the total value was greater than a given amount. The area in the blue rectangle in Columns 1 and 2 is reserved for input data, which are the day name and the value variation for each weekday (no variation occurs on the weekend). The associated program shown in Fig. 1(b) and the spreadsheet formulas in the green rectangle are pre-coded. The associated program is run, upon user request, to eliminate meaningless empty weekend values of Column 2, and to 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.

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

(a) Spreadsheet contents

```
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

In practice, data are filled either manually, or automatically (e.g., copying from somewhere else, or using another associated program `INITIATE`). Then, users 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 may be known only at run-time, whereas the spreadsheet formulas and the associated program are pre-coded.

The final result is computed in the bottom right cell. Its value is **incorrect**. We let $\mathbf{C}[i, j]$ denote the cell in row i and column j. In Fig. 1(a), the cells in 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 whether a cell is empty). Comparison operator ">" always returns **true** when applied to a string and a numeric value, therefore, when cell $\mathbf{C}[i, 5]$ is an empty string, the condition $\mathbf{C}[i, 5] > 150$ evaluates to **true**. Then, built-in function `N` converts **true** into 1. **Therefore, the value produced when evaluating the formula in $\mathbf{C}[45, 5]$ is off by 10**.

This incorrect result is produced without a warning, as it passes through the weak (and incorrect) spreadsheet type checking. Such issues are common in large

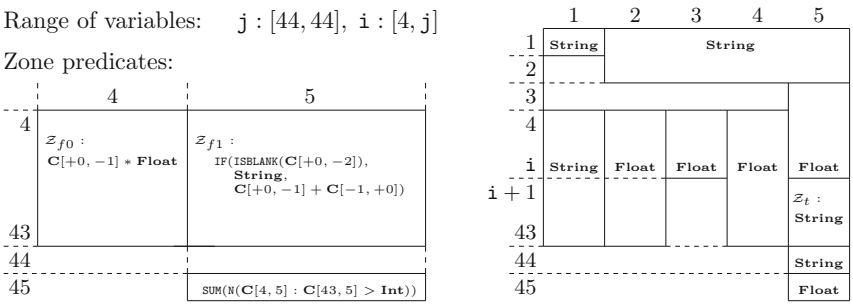


Fig. 2. Abstract state

applications, and hard to diagnose by non-expert users. In particular, testing is likely to miss such problems. In this case, any run with a data sample without empty cells in $\mathbf{C}[4,3] : \mathbf{C}[43,3]$ will produce no incorrect result. Therefore, checking the absence of defects by testing is not possible, especially when input data are made available at run-time, and detecting all such issues will require a *conservative* static analysis that raises a warning whenever an unsafe operation (such as the comparison of a string with a numeric value) might be executed. Different users may consider different sets of operations safe, thus the set of unsafe operations should be a *parameter* of the analysis.

Analysis of the Example. The properties of the application of Fig. 1 are shown in Fig. 2. After Line 14 of the associated program executes, `j` is always equal to 44, whereas `i` may take any value in $[4, 44]$ (as $4 \leq \mathtt{i} \leq \mathtt{j}$). The diagrams show properties that always hold for *zones* in the table: each rectangle accounts for a set of cells, and is labeled by a property of these cells. Cell properties consist either in *abstract formulas* or in *types*. Abstract formulas may use relative indexes (e.g., $\mathbf{C}[+0,-1]$) or absolute indexes (e.g., $\mathbf{C}[4,5]$). The analysis of **Eval** at Line 15 will use abstract formulas to infer type **Float** for zone $\mathcal{Z}_{f0}$ (since the multiplication of *empty* value ϵ and a float value produces float value 0.0 as does the multiplication of two float values), and then split $\mathcal{Z}_{f1}$ into two sub-zones of type **Float** and **String**. We call the latter $\mathcal{Z}'_t$. 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* (reported by the analysis) of a value of type **String** with a value of type **Float**.

Moreover, the analysis should not reject obviously correct applications. For instance, a corrected version of the example application would replace formulas in Column 5 with formulas of the form $\mathtt{IF}(\mathtt{ISBLANK}(\mathbf{C}[4,3]), 0.0, \mathbf{C}[4,4]+\mathbf{C}[3,5])$. Then, all results in Column 5 would have a floating point type, and no unsafe comparison of a string with a numeric value would occur. The same reasoning based on zones will allow to establish this.

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

Fig. 3. Syntax: a core spreadsheet language

3 A Core Spreadsheet Language

In this section, we formalize a core language that incorporates both the spreadsheet table and the runnable code (the analyzer shown in Sections 6 and 7 supports a much wider feature set). This language has several distinctive features. 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*. The value is usually displayed. Cell formulas 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).

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} (resp., Ω_{t}) stands for an execution error (resp., a typing error). We let $\mathbb{X} = \{\mathtt{x}, \mathtt{y}, \ldots\}$ denote a finite set of variables. A variable or a cell content has a type. We assume a set of pre-defined data-types such as not only **Bool**, **Float**, **String**, but also **Date** or **Currency** (which exist in real spreadsheet languages). Moreover, ϵ is the 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 $j \in \mathbb{C}$.

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.). 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 expression value* as in $\mathbf{C}[e_0, e_1] = e_2$ or a *formula* and its currently evaluated value as in $\mathbf{C}[e_0, e_1] = \text{“} = e_2\text{”}$: *unlike an expression, a formula may be re-evaluated in the future.* 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 $\mathbf{C}[2, 4]$. Last, statement **Eval** causes a global re-evaluation of the *all*

formulas in the 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**).

An Excel *spreadsheet application* comprises a 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 a is defined by a single program body s that includes the initialization of the spreadsheet by a series of assignments (and is preceded by the declaration of the variables used in the body of the program): the example of Sect. 2 would be represented by a single 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. 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;   | 4 C[2,1] = " = C[1,1]";  | 7 C[2,2] = " = C[1,1] + 8";
2 x = −5;         | 5 Eval;                  | 8 C[3,2] = " = C[2,1] + C[2,2]"
3 C[1,1] = 6;     | 6 C[1,1] = 24;           |
```

States. At any time in the execution, the memory is defined by the values of variables, and the formulas and values stored in the spreadsheet. Thus, a *non-error 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):

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

$\sigma^{\mathbb{SE}}$:

	1	2
1	$= 24$	
2	$= \mathbf{C}[1,1]$	$= \mathbf{C}[1,1] + 8$
3		$= \mathbf{C}[2,1] + \mathbf{C}[2,2]$

$\sigma^{\mathbb{SV}}$:

	1	2
1	24	ϵ
2	6	32
3	ϵ	38

Semantics of Expressions. The evaluation of an expression e is defined by its semantics $[\![e]\!]_{\mathbb{E}} : \Sigma \to \mathcal{P}(\mathbb{V})$ (note that an expression may evaluate to several values in order to account for possible non-determinism and run-time inputs, which may arise due to calls to `RAND`, `DATE`, or other functions reading 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 $[\![\mathtt{F}]\!] : (\mathcal{P}(\mathbb{V}))^n \to \mathcal{P}(\mathbb{V})$ denote the mathematical function associated with built-in (n-ary) function `F` (note that

their arguments may also be non-deterministic). Then, $[\![e]\!]_{\mathbb{E}}$ can be defined as follows, by induction over the syntax:

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

We remark that this evaluation function uses the last evaluated value whenever it reads a cell. In particular, it does not evaluate the formulas of 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 its descendant $\mathbf{C}[2,1]$ 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. Instead, we are interested in typing errors that may arise when applying an operator or a function to arguments whose types do not match the convention or the expectation of that operator or function. We write Ω_{t} both for the value produced in case of a typing error and for the corresponding error state. 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.

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:

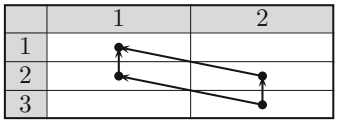
- $[\![s_0;\ s_1]\!]_{\mathbb{P}}(\sigma) = \bigcup\{[\![s_1]\!]_{\mathbb{P}}(\sigma_0) \mid \sigma_0 \in [\![s_0]\!]_{\mathbb{P}}(\sigma)\}$;
- $[\![\textbf{If } e \textbf{ Then } s_0 \textbf{ Else } s_1 \textbf{ End}]\!]_{\mathbb{P}}(\sigma) = S_0 \cup S_1$ where $S_0 = [\![s_0]\!]_{\mathbb{P}}(\sigma)$ 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.

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 two forms of assignments to a cell differ in the fact the formula is preserved *only in the formula assignment*:

- 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 formulas in all the cells of the spreadsheet to be re-evaluated. Therefore, the semantics of **Eval** involves a possibly large number of computation steps, and it boils down to a *fixpoint* computation over the whole spreadsheet that recalculates $\sigma^{\mathbb{SV}}$.

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:



$\sigma^{\mathbb{SV}}_{\text{res}}$:

	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 evaluated yet, and that can be evaluated. To distinguish cells whose value has been calculated 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 $\forall \sigma^{\mathbb{SV}}_0 \in \mathcal{S}_0,\ \exists \sigma^{\mathbb{SV}}_1 \in$

$\mathcal{S}_1,\ \forall(i,j) \in \mathbb{R} \times \mathbb{C},\ \sigma_0^{\mathbb{SV}}(i,j) \sqsubseteq_{\mathbb{V}} \sigma_1^{\mathbb{SV}}(i,j)$. Moreover, we let $\sigma_\perp^{\mathbb{SV}} \in \mathbb{S}_{\mathbb{V}}$ be defined by $\forall(i,j),\ \sigma_\perp^{\mathbb{SV}}(i,j) = \perp$. 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: $\mathbf{lfp}_{\{\sigma_\perp^{\mathbb{SV}}\}}\mathcal{F}_\prec = \mathbf{lfp}_{\{\sigma_\perp^{\mathbb{SV}}\}}\mathcal{F}_{\prec'} = \bigsqcup\{(\mathcal{F}_\prec)^n(\{\sigma_\perp^{\mathbb{SV}}\}) \mid n \in \mathbb{N}\}$. Thus, we define: $[\![\mathbf{Eval}]\!]_{\mathbb{P}}(\sigma) = \{(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma_{\text{res}}^{\mathbb{SV}}) \mid \sigma_{\text{res}}^{\mathbb{SV}} \in \mathbf{lfp}_{\{\sigma_\perp^{\mathbb{SV}}\}}\mathcal{F}_\prec\}$*

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 $\perp$) 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 allows 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). It is based on *abstract formulas* (Sect. 4.1) that summarize the behavior of formulas depending on the type of their inputs and on *abstract zones* [8] that 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. Thus, the effect of formulas should be propagated through the analysis. However, dealing with all formulas stored in the spreadsheet would be too costly. Therefore, we propose an abstraction of the semantics of formulas, which expresses their effect on types, and replaces, e.g., constants with their type:

Definition 1 (Abstract formulas). *Abstract formulas are defined by:*

$$e^\sharp\ (\in \mathbb{E}^\sharp)\ ::=\ t \mid \mathbf{C}[n,n] \mid e^\sharp \oplus e^\sharp \mid \mathtt{F}(e^\sharp, \ldots, e^\sharp) \qquad \text{where } n \in \mathbb{V}_{\text{int}},\ t \in \mathbb{T}$$

Example 2 (Abstract formulas). $\mathbf{Int} + \mathbf{Float}$, $\mathbf{Float} - \mathbf{C}[3,4]$, $\mathtt{ISBLANK}(\mathbf{C}[5,6])$ are all abstract formulas. Moreover, we also allow relative indexes in abstract formulas, as in $\mathbf{C}[+0,-1] \star \mathbf{Float}$ (Fig. 2).

Semantics of Abstract Formulas. We now give a semantics to abstract formulas, following a similar scheme as in Sect. 3, and where abstract formulas evaluate into *types*. 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. 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, comparing a string with an integer is unsafe, so $[\![<]\!]_{\mathrm{t}}(\{\mathbf{String}\},\{\mathbf{Int}\})$ leads to Ω_{t}, as for all unsafe operations. 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:

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

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, and it satisfies the soundness condition: if $e \in \mathbb{E}$, $\sigma^{\mathbb{SV}} \in \mathbb{S}_{\mathbb{V}}$ and $\sigma^{\mathbb{T}} \in \mathbb{S}_{\mathbb{T}}$ are 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 (Formulas abstraction). We have the abstractions $\phi(\mathbf{C}[4,4] * 1.3) = \mathbf{C}[4,3] * \mathbf{Float}$, and with 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 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 **Error**; it always returns a boolean value whatever the argument is, then formula $\mathtt{ISERROR}(\mathbf{C}[5,6])$ can be simplified into **Bool**.

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

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] $\mathbb{D}^\sharp_{\text{num}}$ 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 in 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]. For clarity, we write bounds on $\bar{i}, \bar{j}$ using interval notation and let the zone defined by $\bar{i} \in [e_0, e_1] \wedge \bar{j} \in [e_2, e_3]$ be denoted by $[e_0, e_1] \times [e_2, e_3]$ (where $e_0, \ldots, e_3$ are linear expressions over the variables or constants).

Example 4 (Abstract zones). We define a few zones relevant to the example of Sect. 2. Zone $\mathcal{Z}_0 : \bar{i} \in [4, 43] \wedge \bar{j} = 2$ (or $[4, 43] \times [2, 2]$) describes a block in column 2, from row 4 till row 43. Similarly, zone $\mathcal{Z}_1 : [4, \mathtt{i}] \times [3, 3]$ describes a block in column 4, and spanning from row 4 till row $n_{\mathtt{i}}$, where $n_{\mathtt{i}}$ denotes the value of `i` in the current state. Last, zone $\mathcal{Z}_2 : [4, 43] \times [4, 4]$ describes a block in column 4.

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* $(\mathcal{Z}, \mathcal{P}) \in \mathbb{D}^\sharp_{\text{z}}$*, where* $\mathbb{D}^\sharp_{\text{z}} = \mathbb{D}^\sharp_{\text{num}} \times \mathbb{D}^\sharp_{\text{c}}$*. 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 distinguish zone predicates attached to abstract formulas and zone predicates attached to types: we let $F^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,form}})$ denote the abstract zone predicates for formulas and we let $T^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,type}})$ 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 (Example 4 continued: abstract predicates over zones). The following abstract zone predicates are 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 abstracted by $\mathbf{C}[+0, -1] * \mathbf{Float}$, thus this zone can be described with abstract zone predicate $(\mathcal{Z}_2, \mathbf{C}[+0, -1] * \mathbf{Float})$. 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 an *over*-approximation of the set $[\![a]\!]_{\mathbb{A}}$ or reachable states of an application a, expressed in the abstract domain defined in Sect. 4. 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. We design two *sound* abstract semantics. 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). The abstract semantics $[\![a]\!]^\sharp_{\mathbb{A}} \subseteq \mathbb{D}^\sharp_\Sigma$ of application a is a finite set of abstract states. We defer the analysis of global re-evaluation to Section 5.2 and handle the others first in Section 5.1. Some abstract operations are common with [8] whereas others are deeply different, especially those related to *formulas*.

5.1 Abstract Interpretation of Basic Statements

Straight Line Code. The core language of Sect. 3 features several, rather similar forms of assignments (assignment to a variable, of an evaluated expression to a cell, or of a formula to a cell). Thus, the analysis defines three 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. Information about the overwritten cell should be dropped from the abstract state, either by removing existing zone predicates, or by splitting zones into preserved / overwritten areas. Then, new type and formula information

should be synthesized and attached to a zone corresponding only to the cell overwritten by the assignment. 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 $\sigma^\sharp = (\mathtt{i} < \mathtt{n}, \{([1, \mathtt{i}-1]\times[2,2], e^\sharp), ([\mathtt{i}, \mathtt{n}]\times[2,2], \text{“} = \mathbf{String}\text{”})\})$, where $e^\sharp = \mathbf{Int}+\mathbf{C}[+0,-1]$. Then, assignment $\mathbf{C}[\mathtt{i},2] = \text{“} = 24 + \mathbf{C}[\mathtt{i},1]\text{”}$ 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 $[\mathtt{i},\mathtt{n}] \times [2,2]$ should be removed from that zone. Therefore, we obtain abstract state $\sigma_0^\sharp = (\mathtt{i} < \mathtt{n}, \{([1, \mathtt{i}-1] \times [2,2], e^\sharp), (\mathtt{i} \times [2,2], e^\sharp), ([\mathtt{i}+1, \mathtt{n}] \times [2,2], \text{“} = \mathbf{String}\text{”}), (\mathtt{i} \times [2,2], \mathbf{Int})\})$.

This update operation creates new zones, yet, when several adjacent zones have the same type and abstract formulas, they could be merged, with no loss of information. This operation is performed by an operator $\mathbf{reduce}^\sharp : \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$ introduced in [8], and that satisfies soundness condition $\forall \sigma^\sharp \in \mathbb{D}^\sharp_\Sigma,\ \gamma_\Sigma(\sigma^\sharp) \subseteq \gamma_\Sigma(\mathbf{reduce}^\sharp(\sigma^\sharp))$.

Example 7 (Reduction). In abstract state $\sigma_0^\sharp$ of Example 6, $([1, \mathtt{i}-1] \times [2,2], e^\sharp)$ and $([\mathtt{i}, \mathtt{i}] \times [2,2], e^\sharp)$ can be merged into $([1, \mathtt{i}] \times [2,2], e^\sharp)$.
As $([4,4]\times[4,4], \mathbf{C}[4,3]*\mathbf{Float})$ is equivalent to $([4,4]\times[4,4], \mathbf{C}[+0,-1]*\mathbf{Float})$, and $([5,5]\times[4,4], \mathbf{C}[5,3]*\mathbf{Float})$ is equivalent to $([5,5]\times[4,4], \mathbf{C}[+0,-1]*\mathbf{Float})$, these two zones can be merged into $([4,5] \times [4,4], \mathbf{C}[+0,-1] * \mathbf{Float})$.

We can now define the analysis of straight line code (sequences of assignments):

- $[\![s_0;\ s_1]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = [\![s_1]\!]^\sharp_\mathbb{P}([\![s_0]\!]^\sharp_\mathbb{P}(\sigma^\sharp))$;
- $[\![\mathtt{x} = e]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = \mathbf{reduce}^\sharp(\mathbf{assign}^\sharp_\mathbb{X}(\mathtt{x}, e, \sigma^\sharp))$;
- $[\![\mathbf{C}[e_0, e_1] = e_2]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = \mathbf{reduce}^\sharp(\mathbf{assign}^\sharp_\mathbb{V}(e_0, e_1, e_2, \sigma^\sharp))$;
- $[\![\mathbf{C}[e_0, e_1] = \text{“} = e_2\text{”}]\!]^\sharp_\mathbb{P}(\sigma^\sharp) = \mathbf{reduce}^\sharp(\mathbf{assign}^\sharp_\mathbb{E}(e_0, e_1, e_2, \sigma^\sharp))$.

Control Structures. The analysis of control structures requires condition test, join and widening operators. Condition tests refine information on variable ranges (hence, refining zone bounds) and on cell types (due to operators testing the type of cell values, such as `ISBLANK`). They are analyzed by an operator $\mathbf{guard}^\sharp : \mathbb{E}\times\mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$ that satisfies soundness condition $\forall \sigma \in \gamma_\Sigma(\sigma^\sharp),\ \mathbf{true} \in [\![e]\!]_\mathbb{E}(\sigma) \Rightarrow \sigma \in \gamma_\Sigma(\mathbf{guard}^\sharp(e, \sigma^\sharp))$. Control flow joins are analyzed by a join operator $\sqcup^\sharp : \mathbb{D}^\sharp_\Sigma \times \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$ such that $\forall \sigma_0^\sharp, \sigma_1^\sharp \in \mathbb{D}^\sharp_\Sigma,\ \gamma_\Sigma(\sigma_0^\sharp) \cup \gamma_\Sigma(\sigma_1^\sharp) \subseteq \gamma_\Sigma(\sigma_0^\sharp \sqcup^\sharp \sigma_1^\sharp)$, whereas loops require a widening operator $\nabla^\sharp$, based on similar algorithms and that ensures the termination of abstract iterates. These operators *generalize* bounds on zones [8], hence play a critical role in the inference of non trivial zone invariants, such as those shown in Fig. 2:

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

We can now define the analysis of condition statements and loops:

– $[\![\mathbf{If}\ e\ \mathbf{Then}\ s_0\ \mathbf{Else}\ s_1\ \mathbf{End}]\!]^\sharp_{\mathbb{P}}(\sigma^\sharp) = [\![s_0]\!]^\sharp_{\mathbb{P}}(\mathbf{reduce}^\sharp(\mathbf{guard}^\sharp(e, \sigma^\sharp))) \sqcup^\sharp [\![s_1]\!]^\sharp_{\mathbb{P}}(\mathbf{reduce}^\sharp(\mathbf{guard}^\sharp(\neg e, \sigma^\sharp)))$;

– $[\![\mathbf{While}(e)\ s\ \mathbf{End}]\!]^\sharp_{\mathbb{P}}(\sigma^\sharp) = \mathbf{reduce}^\sharp(\mathbf{guard}^\sharp(\neg e, \mathbf{lfp}^\sharp_\bot F^\sharp))$ where $F^\sharp(\sigma_0^\sharp) = \sigma^\sharp \sqcup^\sharp [\![s]\!]^\sharp_{\mathbb{P}}(\mathbf{reduce}^\sharp(\mathbf{guard}^\sharp(e, \sigma_0^\sharp)))$ and $\mathbf{lfp}^\sharp$ computes abstract post-fixpoint, using classical abstract iteration techniques, using widening operator $\nabla^\sharp$.

We recall the abstract post-fixpoint operator is sound in the following sense: if $F : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ is continuous and $F^\sharp : \mathbb{D}^\sharp_\Sigma \to \mathbb{D}^\sharp_\Sigma$, $\mathcal{S} \subseteq \Sigma$ and $\sigma^\sharp \in \mathbb{D}^\sharp_\Sigma$ are such that, $F \circ \gamma_\Sigma \subseteq \gamma_\Sigma \circ F^\sharp$ and $\mathcal{S} \subseteq \gamma_\Sigma(\sigma^\sharp)$, then $\mathbf{lfp}_\mathcal{S} F \subseteq \gamma_\Sigma(\mathbf{lfp}^\sharp_{\sigma^\sharp} F^\sharp)$.

Applications. The analysis of an application a recursively computes the abstract semantics of all statements in the body of a from its initial state, and produces a finite set of abstract states $[\![a]\!]^\sharp_{\mathbb{A}}$. Our analysis is sound:

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.

5.2 Abstract Interpretation of Global Evaluation

The concrete semantics of **Eval** boils down to a fixpoint, that re-computes cell values in *the whole* spreadsheet while preserving formulas and variables values (Sect. 3). Thus, we assume $\sigma^\sharp = (N^\sharp, F^\sharp \uplus T^\sharp)$, and show the computation of $T^\sharp_{\text{res}}$ (by fixpoint approximation) so as to let $[\![\mathbf{Eval}]\!]^\sharp(\sigma^\sharp) = (N^\sharp, F^\sharp \uplus T^\sharp_{\text{res}})$. We first show a very basic iteration strategy, and then discuss the analysis of **Eval**.

Cell-by-Cell Re-evaluation. The concrete semantics of **Eval** is based on function $\mathcal{F}_\prec$, defined by a cell ordering $\prec$ compatible with formula dependencies. In this paragraph, we show an abstract counterpart for $\mathcal{F}_\prec$ under the assumption that each abstract zone is reduced to a single concrete cell, thus elements of $T^\sharp$ (resp., $F^\sharp$) are equivalent to functions from $\mathbb{R} \times \mathbb{C}$ into $\mathbb{T}$ (resp., $\mathbb{E}^\sharp$). Abstract formulas follow the same dependencies as concrete formulas, thus the topological order $\prec$ can be retrieved from an abstract state, by topological sorting. Moreover, the analysis should support "not yet re-evaluated" cells, which are denoted by $\bot$:

– To extend type spreadsheets $\mathbb{S}_\mathbb{T}$, we 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 $\mathbb{S}_{\mathbb{T}\perp}$.
- We let $T^\sharp_\perp \in \mathbb{S}_{\mathbb{T}\perp}$ be defined by $\forall(i,j),\ T^\sharp_\perp(i,j) = \perp$.
- Given an abstract formula $e^\sharp$, its 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.

We can now define the abstract counterpart $\mathcal{F}^\sharp_\prec : \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,type}}) \to \mathcal{P}_{\text{fin}}(\mathbb{D}^\sharp_{\text{z,type}})$ of $\mathcal{F}_\prec$. It is such that for all $T^\sharp_0 \in \mathbb{S}_{\mathbb{T}\perp}$, and for all i,j, $\mathcal{F}^\sharp_\prec(T^\sharp_0)(i,j) = [\![F^\sharp(i,j)]\!]_{\text{t}}(T^\sharp_0)$ when $T^\sharp_0(i,j) = \perp$ and $\forall(i',j') \prec (i,j),\ T^\sharp_0(i',j') \neq \perp$ (otherwise $\mathcal{F}^\sharp_\prec(T^\sharp_0)(i,j) = T^\sharp_0(i,j)$). It is sound: for all $\sigma^\sharp_0 = (N^\sharp, F^\sharp \uplus T^\sharp_0) \in \mathbb{D}^\sharp_\Sigma$, and for all $(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \sigma^{\mathbb{SV}}_0) \in \gamma_\Sigma(\sigma^\sharp_0)$, we have $(\sigma^{\mathbb{X}}, \sigma^{\mathbb{SE}}, \mathcal{F}_\prec(\sigma^{\mathbb{SV}}_0)) \subseteq \gamma_\Sigma(N^\sharp, F^\sharp \uplus \mathcal{F}^\sharp_\prec(T^\sharp_0))$. 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* $T^\sharp_{\text{res}} = \mathbf{lfp}^\sharp_{T^\sharp_\perp} \mathcal{F}^\sharp_\prec = \bigsqcup\{(\mathcal{F}^\sharp_\prec)^n(T^\sharp_\perp) \mid n \in \mathbb{N}\}$ *defines a sound post-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**:

	1	2
1	= **Int**	= **Empty**
2	= **C**[1, 1]	= **C**[1, 1] + **Int**
3	= **Empty**	= **C**[2, 1] + **C**[2, 2]

	1	2
1	**Int**	(**Empty**)
2	**Int**	**Int**
3	(**Empty**)	**Int**

Zone-by-Zone Strategy. The cell-by-cell strategy abstract interpretation of **Eval** would not be efficient in practice, as abstract states usually contain zones which are bounded, but possibly large and/or of variable size. However, since a whole zone is attached to a single abstract formula, type information for a whole zone can often be computed in a single step, which is much faster than cell-by-cell evaluation.

A zone can be re-evaluated as soon as the two following conditions are satisfied: (1) its abstract formulas induce no internal dependency, i.e., between its cells (in the example of Sect. 2, this holds for all zones, except 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.

When a zone satisfies these two conditions, the analysis can re-evaluate its type by applying the abstract formula it corresponds to, since its arguments have already been re-evaluated. When an argument of the abstract formula may belong to several zones, it will be necessary to split the zone being re-evaluated. This will produce a set of zones with type information. The analysis will apply this efficient scheme whenever the topological ordering induced by abstract formulas zones allows it:

Example 10 (Zone-by-zone strategy). We assume the following abstract state:

	1	2	3	4
1	$\mathcal{Z}_0$	$\mathcal{Z}_1$	$\mathcal{Z}_2$	$\mathcal{Z}_3$
i	$= \mathbf{Float}$	$= \mathbf{C}[+0,-1] + \mathbf{Int}$	$= \mathbf{C}[+0,-2] * \mathbf{Float}$	$= \mathbf{C}[+0,-2] < \mathbf{C}[+0,-1]$

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

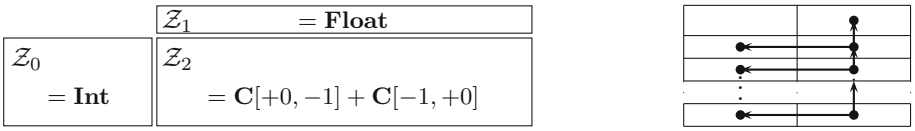
Abstract Iteration over Zones, Using Widening. When a zone contains internal dependencies (i.e., abstract formula using as arguments cell that belong to the zone itself), the zone-by-zone strategy does not apply. Such a *self-reference* occurs in the example of Sect. 2 since the evaluation of Column 5 requires types of Columns 3, 4 and Column 5 itself. *Inter-reference* among zones may also occur, e.g., when $\mathcal{Z}_0$ needs types of $\mathcal{Z}_1$, $\mathcal{Z}_1$ needs types of $\mathcal{Z}_2$, ..., and $\mathcal{Z}_n$ needs types of $\mathcal{Z}_0$.

Zones containing such patterns can be re-evaluated in the abstract level by simulating a cell-by-cell re-evaluation order, as part of an abstract fixpoint computation. To do this, under the assumption that there is no cycle in formulas (this case is discussed in Sect. 6), the analysis of **Eval** will consider a loop that computes the cells in the zone one-by-one, following the steps below:

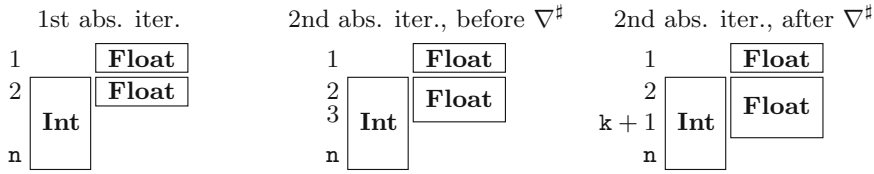
1. introducing loop variable `k`, denoting the number of cells in the abstract formula zone that have been re-evaluated;
2. determining the first cell in the abstract formula zone dependency order;
3. splitting the abstract formula zones into two zones, respectively for cells that can be immediately re-evaluated, and for cells that cannot be re-evaluated yet;
4. iterating Steps 2 and 3 until the abstract formula zones are fully treated, and applying widening at each step to ensure termination, thanks to $\mathbf{lfp}^\sharp$;
5. synthesizing the final abstract state by restricting, when Step 4 produces 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:



On the third iteration, abstract states are stable. Moreover, the analysis proves all the formulas 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: it will always try to detect and to re-compute first the zones that can be evaluated as a whole. This strategy is the most efficient and turns out to be the most frequently used in practice. Remaining cases are dealt with by widening based and cell-by-cell strategies. After adding **Eval** statement to the set of the statements, the global soundness theorem (Theorem 2) still holds.

6 Implementation of an Excel VBA Analyzer

We have implemented our analysis. Our analyzer handles a large subset of Microsoft Excel functions and VBA, following the VBA specification [2]. Our tool consists of a frontend written in VBA, that parses Microsoft Excel spreadsheet tables (e.g., number formats, types, formulas, buttons) and VBA macros, and exports them to the static analyzer itself, which undertakes the verification (and includes 19000 lines of OCaml code). 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 formalized the analysis of the core spreadsheet language, but our analyzer supports many additional spreadsheet features, to be able to cope with real-world applications, including the following:

- A *workbook* may contain several worksheets, and formulas may refer to cells in another sheet or another workbook.
- Macros may contain *interprocedural calls*, other user-defined subroutine or function, 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 in the initial abstract state. This allows a precise verification of spreadsheets that utilize data unknown at verification time / non-deterministic 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 analysis warnings, and adopt the analysis ?

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 under development and are already operational.

A spreadsheet may contain zero, one or several macros. It may also not contain any formula. 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 rest 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. The analyzer was set up in such a way that, when an unsafe typing rule is applied, it raises an alarm and stops the analysis. Therefore, the number of alarms raised corresponds to the number of spreadsheets / macros in the USES Corpus 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 found formula "=`AVERAGE`(D4;F4;H4;J4;L40)", was referring to "L40" although it was 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 considered the formula valid. This mistake will indeed result in the computation of an incorrect average grade.

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

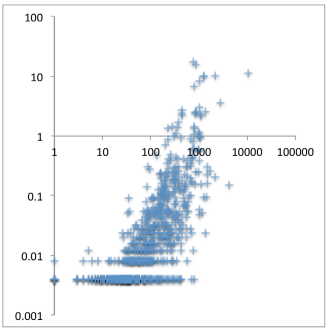
inappropriate manipulation (e.g., mistyping, improper copy-paste). In total, we found 13 bugs of this category. (2) Formulas or statements have a certain assumption for the types or the values of input data, yet the assumption is not specified or will not always hold at run-time. For instance, in a macro 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.

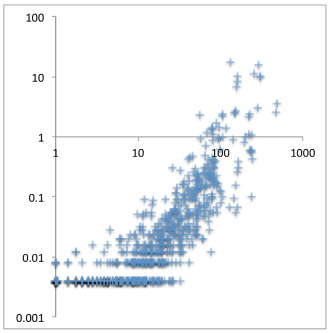
Analysis Time. The analyzer succeeded in verifying 858 macros in 161 spreadsheets (Categories B + C in Table 7). The size of each macro ranges from a few LOCs to several hundred LOCs. As one LOC could well involve a complex abstract operation by executing a complex statement or calling another macro, 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 [3,9,4,5,6] 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) we consider classical types in the programming language point of view, whereas their types refer to the concrete meaning of objects; thus, the built-in rules or bugs discovered by the two analyses are different; (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 [28]. One difference of our work is that we treat bi-dimensional arrays, whereas the existing works study 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, without formulas that can be re-evaluated after their inputs change. We aim at verifying real-world spreadsheets, which consist of associated programs *and* formulas. To this end, we formalize a larger spreadsheet language which includes formulas, and propose an abstraction that ties not only types but also abstract formulas to zones. Therefore, unlike [8], our analysis can cope with the re-evaluation of formulas (in automatic mode, upon user-request or from the associated programs), which is critical to handle real-world spreadsheets. Last, we evaluate the analysis and the implementation by analyzing a large set of real-world spreadsheets.

JavaScript, and Languages with Dynamic Evaluation. Thiemann [26] defines a type system that flags suspicious type conversions in JavaScript programs, which is a similar verification target to ours, albeit for a different language. 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. However, spreadsheet languages distinguish themselves from other scripting and dynamic languages by the way dynamicity is implemented: formulas are structured and organized in a two-dimensional array, whereas the `eval` function in JavaScript applies to strings and has a very different semantics. This led us to a very different abstraction based on zone and abstract formulas, than that of [18]. Moreover, 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 (January 2013)

2. MS-VBAL: VBA language specification. Tech. rep., Microsoft Corporation (April 2014)
3. Abraham, R., Erwig, M.: Header and unit inference for spreadsheets through spatial analyses. In: Visual Languages and Human-Centric Computing. 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: International Conference on Software Engineering (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: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 94–110. Springer, Heidelberg (2012)
9. Coblenz, M.J., Ko, A.J., Myers, B.A.: Using objects of measurement to detect spreadsheet errors. In: Visual Languages and Human-Centric Computing (2005)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages. ACM (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages. ACM (1979)
12. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Principles of Programming Languages. ACM (2011)
13. Cunha, J., Fernandes, J.P., Ribeiro, H., Saraiva, J.: Towards a catalog of spreadsheet smells. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part IV. LNCS, vol. 7336, pp. 202–216. Springer, Heidelberg (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: Workshop on End-User Software Engineering (2005)
16. Hammer, M.A., Phang, K.Y., Hicks, M., Foster, J.S.: Adapton: Composable, demand-driven incremental computation. In: Programming Language Design and Implementation. ACM (2014)
17. Hermans, F., Pinzger, M., Deursen, A.V.: Detecting and visualizing inter-worksheet smells in spreadsheets. In: International Conference on Software Engineering (2012)
18. Jensen, S.H., Jonsson, P.A., Møller, A.: Remedying the eval that men do. In: International Symposium on Software Testing and Analysis. ACM (2012)
19. Jones, S.P., Blackwell, A., Burnett, M.: A user-centred approach to functions in Excel. In: International Conference on Functional Programming. ACM (2003)
20. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation (2006)
21. Panko, R.R.: What we know about spreadsheet errors. Journal of End User Computing (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: Principles of Programming Languages. 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. Thiemann, P.: Towards a type system for analyzing javascript programs. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 408–422. Springer, Heidelberg (2005)
27. Wakeling, D.: Spreadsheet functional programming. Journal of Functional Programming (2007)
28. Xi, H., Pfenning, F.: Eliminating array bound checking hrough dependent types. In: Programming Language Design and Implementation. ACM (1998)