An Abstract Domain to Infer Types over Zones in Spreadsheets

Tie Cheng, Xavier Rival

An Abstract Domain to Infer Types over Zones in Spreadsheets [⋆]

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. Spreadsheet languages are very commonly used, by large user bases, yet they are error prone. However, many semantic issues and errors could be avoided by enforcing a stricter type discipline. As declaring and specifying type information would represent a prohibitive amount of work for users, we propose an abstract interpretation based static analysis for spreadsheet programs that infers type constraints over zones of spreadsheets, viewed as two-dimensional arrays. Our abstract domain consists in a cardinal power from a numerical abstraction describing zones in a spreadsheet to an abstraction of cell values, including type properties. We formalize this abstract domain and its operators (transfer functions, join, widening and reduction) as well as a static analysis for a simplified spreadsheet language. Last, we propose a representation for abstract values and present an implementation of our analysis.

1 Introduction

Spreadsheet softwares such as Excel or OpenOffice are very widely used, and include not only an interface to visualize and manipulate two-dimensional arrays of cells but also a programming language which permits complex calculations. For instance, Excel includes Visual Basic for Applications (VBA) and OpenOffice includes a Basic like language.

These programming languages are used in many industrial and financial areas for important applications such as statistics, organization and management. Reports of spreadsheet related errors appear in the global media at a fairly consistent rate. It is not surprising that, as an example, a consulting firm, Coopers and Lybrand in England, found that 90% of all spreadsheets with more than 150 rows that it audited contained errors [1]. Spreadsheet errors result in various problems such as additional audit costs, money loss, false information to public, wrong decision making, etc. As the risks they incur are not considered acceptable, the defects in such applications have attracted increasing attention from communities such as Excel advanced users and IT professionals.

[⋆] 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.

Various techniques were considered in order to tackle spreadsheet risks. One class of existing work enhances the functional aspect of spreadsheets, viewed as a first-order functional environment [2,3,4,5]. Another body of work attempts to improving quality of spreadsheets using model-driven engineering spreadsheet development environments [6,7,8,9]. Last, ad hoc methods [10,11,12] were proposed to detect specific kinds of problems, using most of the time algorithms with no mathematical foundation, that neither sound nor complete. One major drawback of the existing work is that currently they only consider spreadsheet interface, but not consider applications attached to the spreadsheets, which are written e.g., in VBA.

In this paper, we address the lack of static types in current spreadsheet applications. For instance, it is impossible to declare abstract types (e.g. integer, boolean, etc.) for a cell in Microsoft Excel; a value of any type may be assigned to any cell at any time. This feature of spreadsheet applications may provide users with a high level of flexibility but it becomes a serious source of errors which would be avoided in well typed languages.

Therefore, we verify the absence of some class of type errors to improve the safety of spreadsheet programs, that existing research in enhancing spreadsheet languages or focusing on spreadsheet interface hardly deals with. Our approach is based on a static analysis by abstract interpretation, which guarantees the soundness of our approach and makes it possible to existing abstract domains and tools.

More precisely, our analysis aims at inferring type information about zones in spreadsheets, taking into account an initial condition on the spreadsheet, and all possible sequences of operations of the associated programs. We make the following contributions:

- we introduce an abstract domain to express type properties of array zones, based on a cardinal power of zone abstractions and type abstractions (Sect. 4);
- we propose a set of transfer functions and join and widening operators for the analysis of spreadsheet programs (Sect. 5);
- we validate our approach using a prototype implementation (Sect. 6) with the analysis of simple spreadsheet programs.

2 Overview

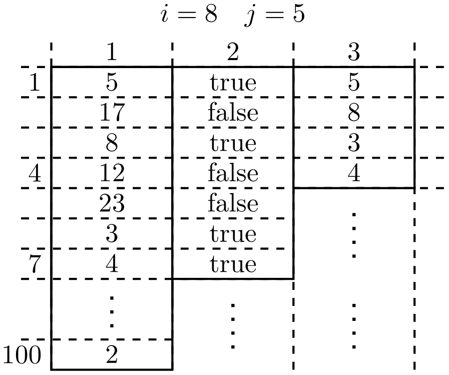
We show a simple program 1 in a restricted spreadsheet language that we consider in the paper. Although its syntax is not exactly the same as that of VBA or Basic in OpenOffice, its task of selecting data sharing certain properties is realistic and common in practice. The rectangle zone $[1, 100] \times [1, 1]$ of the sheet has already been initialized to integer values. The main procedure goes through column 1, for each cell, we compare its value to 9 and assign the boolean result to the cell in the same line in column 2. If the integer value is less than 9, it is copied to the first empty cell in column 3. Our analyzer infers invariants about types of different parts of the sheet, and it also detects different sorts of type conflicts hidden in the code, which may result in strange behaviors in VBA for

```
1  program
2  var
3     i, j : int;
4  name
5     ([1, 100], [1, 1]) : int;
6  begin
7     i := 1; j := 1;
8     while i < 51 do
9        Sheet(i, 2) := Sheet(i, 1) < 9;
10       if Sheet(i, 2) then begin
11          Sheet(j, 3) := Sheet(i, 1);
12          j := j + 1
13    end fi
14    i := i + 1
15 od;
16 Sheet(true, 1) := 20;
17 Sheet(9, 9) := 1 + Sheet(i − 1, 2);
18 if Sheet(j − 1, 3) then
19    Sheet(9, 9) := 1 else
20    Sheet(9, 9) := 2 fi
21 Sheet(i, 1) := true
22 end.
```

Fig. 1: An example program

instance. The index error in line 16, operand error in line 17, condition error in line 18 and assignment error in line 21 will be more explained further.

The concrete states refer to the concrete value of variables and the run-time contents of the sheet cells. Figure 2 represents a concrete state of loop between line 8 and line 9. Precisely, the cells $[1, 7] \times [2, 2]$ store boolean values, the cells $[1, 4] \times [3, 3]$ store integer values. $[1, 100] \times [1, 1]$ is reserved by the declaration **name** at lines 4 and 5 of the program, which means only integer values should be stored in that area.



$i = 8 \quad j = 5$

	1	2	3
1	5	true	5
	17	false	8
	8	true	3
4	12	false	4
	23	false	⋮
	3	true	
7	4	true	
	⋮	⋮	⋮
100	2		

Fig. 2: A concrete state

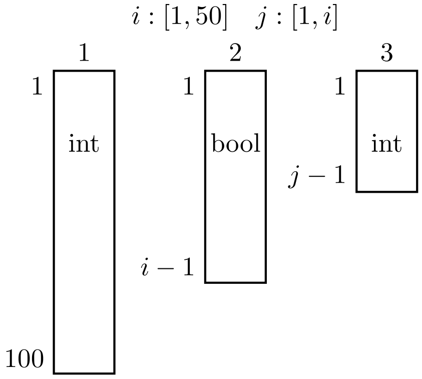


Fig. 3: An abstract state

In order to verify that no illegal operation will be performed in the spreadsheet program due to type issue, and that type properties in the reserved zone are not violated, the analysis should relate type information to spreadsheet cells. Attaching a type predicate to each cell would not be effective, and it would not be even doable for spreadsheet regions of non fixed size. Moreover, users naturally view a spreadsheet as a set of zones where they store homogeneous sorts of data. Therefore, we consider an abstraction, where type information are attached to spreadsheet zones. Precisely, an abstract state will consist in a pair made of an abstraction of integer variables, and an abstraction of type properties of sheet cells (e.g.. Figure 3 is an intuitive image of the abstract state corresponding to the concrete state of Figure 2). The abstraction of integer variables relies on inequality constraints over variables and constants. The abstraction of type properties of sheet cells consists in a form of cardinal power [13] from the abstraction of sheet zones to type properties, where zones are characterized by set of constraints tying cell coordinates to program variables. Although analyzing the simple program, chosen to facilitate the illustration, needs only column abstraction, the abstraction of a large variety of zones is necessary in practice and can be achieved by our domain (e.g. rectangular abstraction is performed to analyze programs in Sect. 6).

3 Spreadsheet Programs

In this paper, we focus on a restricted spreadsheet language, where instructions are assignments, "if" and "while" statements, with data-types ranging in $\mathbb{T} = \{\text{int}, \text{bool}, \ldots\}$. We assume program variables all have type int, whereas spreadsheet cells may store values of any type. Compared to a classical imperative language, our language has two specific features. First, the keyword "**Sheet**" denotes a cell in the spreadsheet. For instance, expression "$\textbf{Sheet}(br, bc)$" evaluates into the value stored in cell (br, bc); "$\textbf{Sheet}(br, bc) := e$" affects the value of e to the cell (br, bc). In the paper, we restrict to cell indexes (br and bc) which are either an integer constant c or an integer variable plus an integer constant $x + c$. Other variables are assumed to be declared at the beginning of the program. Second, spreadsheet areas can be reserved to a type by the keyword "**name**". For instance, "$\textbf{name}\ ([1, 100], [1, 1]) : \text{int}$" in Program 1 means that only integer values should be stored in that area (storing a value of another type would be considered a semantic error).

In the following, we let $\mathbb{V}$ (resp., $\mathbb{V}_{\mathbf{i}}$) denote the set of values (resp., integer values). We let $\mathbb{X}$ denote the set of program variables, augmented with two special variables $\overline{x}, \overline{y}$ which we will use to express relations over cell indexes. Moreover, $\mathbb{N}^2$ represents the set of cells. We use an operational semantics, which collects all program executions. An execution is a sequence of states (or trace), where a state is a pair made of a control state l and a memory state $\rho = (\rho_v, \rho_s)$, where $\rho_v : \mathbb{X} \to \mathbb{V}_{\mathbf{i}}$ and $\rho_s : \mathbb{N}^2 \to \mathbb{V}$ are respectively functions mapping integer variables and sheet cells into values. We let $\to$ denote the transition relation from one state to another (modelling one step of computation) and Ω represent the error state (no transition from Ω is possible). For a detailed presentation of the syntax and concrete semantics of our restricted spreadsheet language, see [14].

4 Abstract Domain

In this section, we formalize the abstract domain used in our analysis as a cardinal power. First, we consider in Sect. 4.1 the abstraction of a set of spreadsheets using one type and constraints of one zone. Then, we show the case of a set of zones in Sect. 4.2. Last, we specialize our abstraction using Difference-Bound Matrices (DBMs) as a base abstraction in Sect. 4.3.

4.1 Abstraction of a Typed Zone

We assume a numerical abstract domain $\mathbb{D}_{\mathrm{i}}^{\sharp}$ is fixed for the abstraction of numerical variables, with a concretization function $\gamma_{\mathrm{i}} : \mathbb{D}_{\mathrm{i}}^{\sharp} \to \mathcal{P}(\mathbb{X} \to \mathbb{V}_{\mathbf{i}})$.

Abstraction of a typed zone. An abstract value in the typed zone abstract domain $\mathbb{D}_{\mathrm{z},1}^{\sharp}$ consists in a pair $(\mathcal{Z}, t)$ where $\mathcal{Z} \in \mathbb{D}_{\mathrm{i}}^{\sharp}$ describes a set of numerical constraints (binding $\overline{x}, \overline{y}$ to other variables and constants in the store) and t is a data-type. The meaning of such an abstract value is that all cells the coordinates $(\overline{x}, \overline{y})$ of which satisfy constraints $\mathcal{Z}$ store a value of type t. More formally, this yields the concretization relation below:

$$\gamma_{\mathrm{z},1}(\mathcal{Z}, t) \stackrel{\triangle}{=} \{(\rho_v, \rho_s) \mid \forall x, y \in \mathbb{N}^2, \rho_v \in \gamma_{\mathrm{i}}(\mathcal{Z}|_{\overline{x}=x, \overline{y}=y}) \Rightarrow \rho_s(x, y) : t\}$$

The concrete state shown in Figure 2 can be approximated by the following typed zone abstract elements:

- $(\mathcal{Z}_0, t_0)$ where $\mathcal{Z}_0 = 1 \leq \overline{x} \wedge \overline{x} \leq 100 \wedge \overline{y} = 1$ and $t_0 = \mathrm{int}$
- $(\mathcal{Z}_1, t_1)$ where $\mathcal{Z}_1 = 1 \leq \overline{x} \wedge \overline{x} \leq i - 1 \wedge \overline{y} = 2$ and $t_1 = \mathrm{bool}$
- $(\mathcal{Z}_2, t_2)$ where $\mathcal{Z}_2 = 1 \leq \overline{x} \wedge \overline{x} \leq j - 1 \wedge \overline{y} = 3$ and $t_2 = \mathrm{int}$

This construction is an abstraction of the cardinal power [13]. Indeed the cardinal power abstract domain would collect all monotone functions from an abstraction of zones into a type domain. We perform here an additional step of abstraction, where functions from zones to types are approximated with only one pair leaving the other zones unconstrained.

Product abstraction. In practice, we always consider an abstraction over the variables together with an abstraction of the spreadsheet contents, using a product domain $\mathbb{D}_{\times,1}^{\sharp} = \mathbb{D}_{\mathrm{i}}^{\sharp} \times \mathbb{D}_{\mathrm{z},1}^{\sharp}$. An abstract value consists in a pair $(\mathcal{V}, \{(\mathcal{Z}, t)\})$ where $\mathcal{V} \in \mathbb{D}_{\mathrm{i}}^{\sharp}$ describes constraints over variables and $(\mathcal{Z}, t) \in \mathbb{D}_{\mathrm{z},1}^{\sharp}$ describes constraints over one zone and its type. Therefore, the combined domain concretization boils down to

$$\gamma_{\times,1}(\mathcal{V}, \{(\mathcal{Z}, t)\}) \stackrel{\triangle}{=} \{(\rho_v, \rho_s) \mid \rho_v \in \gamma_{\mathrm{i}}(\mathcal{V}) \wedge (\rho_v, \rho_s) \in \gamma_{\mathrm{z},1}(\mathcal{Z}, t)\}$$

As an example, the concrete state shown in Figure 2 can be approximated by abstract state $(\mathcal{V}, \{(\mathcal{Z}, t)\}))$ where $\mathcal{V} = 1 \leq i \wedge i \leq 50 \wedge 1 \leq j \wedge j \leq i$, $\mathcal{Z} = 1 \leq \overline{x} \wedge \overline{x} \leq 100 \wedge \overline{y} = 1$ and $t = \mathrm{int}$. We will consider the case of a combined abstraction with several typed zones in Sect. 4.2, after studying some properties of the product abstraction.

Properties. The definition of $\gamma_{\mathrm{z},1}$ and $\gamma_{\times}$ allows to prove the properties below:

1. Propagating constraints over variables into the zone abstraction preserves concretization: $\gamma_{\times}(\mathcal{V}, \{(\mathcal{Z}, t)\}) = \gamma_{\times}(\mathcal{V}, \{(\mathcal{Z} \sqcap \mathcal{V}, t)\})$, where $\sqcap$ simply joins two sets of constraints.

2. Replacing the abstraction of variables (resp. zones) with an equivalent abstraction preserves concretization: if $\gamma_{\mathrm{i}}(\mathcal{V}) = \gamma_{\mathrm{i}}(\mathcal{V}') \wedge \gamma_{\mathrm{i}}(\mathcal{Z}) = \gamma_{\mathrm{i}}(\mathcal{Z}')$ then $\gamma_{\times}(\mathcal{V}, \{(\mathcal{Z}, t)\}) = \gamma_{\times}(\mathcal{V}', \{(\mathcal{Z}', t)\})$

3. Replacing the abstraction of variables with a weaker abstraction results in a weaker abstract state: if $\gamma_{\mathrm{i}}(\mathcal{V}) \subseteq \gamma_{\mathrm{i}}(\mathcal{V}')$ then $\gamma_{\times}(\mathcal{V}, \{(\mathcal{Z}, t)\}) \subseteq \gamma_{\times}(\mathcal{V}', \{(\mathcal{Z}, t)\})$

4. Replacing the zone abstraction with a weaker abstraction results in a stronger abstract state: if $\gamma_{\mathrm{i}}(\mathcal{Z}) \subseteq \gamma_{\mathrm{i}}(\mathcal{Z}')$ then $\gamma_{\times}(\mathcal{V}, \{(\mathcal{Z}, t)\}) \supseteq \gamma_{\times}(\mathcal{V}, \{(\mathcal{Z}', t)\})$

4.2 Abstraction of a Set of Typed Zones

In practice, we need to bind several distinct zones in the spreadsheet to type information. For instance, three zones are needed to faithfully abstract the concrete state of Figure 2. Therefore, we define $\mathbb{D}^{\sharp}_{\mathbf{Z}}$ as the set of finite sets of elements of $\mathbb{D}^{\sharp}_{\mathrm{z},1}$, with concretization $\gamma_{\mathbf{Z}}$ defined by:

$$\gamma_{\mathbf{Z}}(\{(\mathcal{Z}_0, t_0), \ldots, (\mathcal{Z}_n, t_n)\}) \stackrel{\triangle}{=} \bigcap_{0 \leq k \leq n} \gamma_{\mathrm{z},1}(\mathcal{Z}_k, t_k)$$

The definition of the product domain given in Sect. 4.1 extends in a straightforward manner, and the properties mentioned in Sect. 4.1 still hold:

$$\begin{aligned}&\gamma_{\times}(\mathcal{V}, \{(\mathcal{Z}_0, t_0), \ldots, (\mathcal{Z}_n, t_n)\})\\ &\stackrel{\triangle}{=} \{(\rho_v, \rho_s) \mid \rho_v \in \gamma_{\mathrm{i}}(\mathcal{V}) \wedge (\rho_v, \rho_s) \in \gamma_{\mathbf{Z}}(\{(\mathcal{Z}_0, t_0), \ldots, (\mathcal{Z}_n, t_n)\})\}\end{aligned}$$

Then, the concrete state of Figure 2 can be described by the abstract state $(\mathcal{V}, \{(\mathcal{Z}_0, t_0), (\mathcal{Z}_1, t_1), (\mathcal{Z}_2, t_2)\})$ with the notations used in Sect. 4.1. This abstract state actually corresponds to Figure 3.

4.3 An Instantiation with Difference-Bound Matrices

When abstracting array properties, bounds of the form c or $x + c$ are often expressive enough to capture large classes of invariants. Similarly, we found that such bounds are usually adequate to describe spreadsheet zones. This suggests using an abstraction based on Difference-Bound Matrices (DBM) (a weaker form of octagons [15], where constraints are either of the form $c \leq x$, $x \leq c$ or $x - y \leq c$) in order to describe zones. We actually do not need full expressiveness of DBMs in order to describe zones, as we will be interested only in relations that relate an index variable ($\overline{x}$ or $\overline{y}$) to a constant or an expression of the form $x + c$. Therefore, in the following, we set the following abstraction:

- program variables abstractions ($\mathcal{V}$) are described by DBMs;
- zones abstractions ($\mathcal{Z}$) are described by a weaker form of DBMs, where no relation among pairs of variables $u, v \notin \{\overline{x}, \overline{y}\}$ is represented.

A large variety of zones can be expressed using this abstraction, including in particular rectangular ($c_0 \leq \overline{x} \leq c_1, c_2 \leq \overline{y} \leq c_3$), triangular ($c_0 \leq \overline{x} \leq \overline{y}, c_0 \leq \overline{y} \leq c_3$), and trapezoidal ($c_0 \leq \overline{x} \leq \overline{y} + c_1, c_2 \leq \overline{y} \leq c_3$) zones. As shown in Sect. 4.1, this set of constraints allows us to describe all zones relevant in the example program of Fig. 1.

In this step, the classical representation of DBMs using matrices of difference appears unnecessarily heavy for zone constraints $\mathcal{Z}_p$, as no relation needs to be stored for pairs of program variables in $\mathcal{Z}_p$. This leads us to a hollow representation of the $\mathcal{Z}_p$ DBMs, where the submatrix corresponding to the integer variables is removed. We call this representation "Matrix Minus Matrix" (or MMM).

For instance, letting $\mathbf{d}$ (resp., $\mathbf{m}$) denote a DBM (resp., MMM) in the following, all concrete states at the beginning of line 9 in Program 1 can be over-approximated by the abstract value $(\mathbf{d}, \{(\mathbf{m}_0, \mathrm{int}), (\mathbf{m}_1, \mathrm{bool}), (\mathbf{m}_2, \mathrm{int})\})$ (depicted in Figure 3), where

$$\mathbf{d} = \begin{array}{c|ccc} & i & j & 0 \\ \hline i & 0 & 0 & -1 \\ j & +\infty & 0 & -1 \\ 0 & 50 & +\infty & 0 \end{array} \qquad \mathbf{m}_0 = \begin{array}{c|cc|ccc} & \overline{x} & \overline{y} & i & j & 0 \\ \hline \overline{x} & 0 & +\infty & +\infty & +\infty & -1 \\ \overline{y} & +\infty & 0 & +\infty & +\infty & -1 \\ \cline{4-6} i & +\infty & +\infty & & & \\ j & +\infty & +\infty & & & \\ 0 & 100 & 1 & & & \end{array} \qquad \mathbf{m}_1 = \begin{array}{c|cc|ccc} & \overline{x} & \overline{y} & i & j & 0 \\ \hline \overline{x} & 0 & +\infty & +\infty & +\infty & -1 \\ \overline{y} & +\infty & 0 & +\infty & +\infty & -2 \\ \cline{4-6} i & -1 & +\infty & & & \\ j & +\infty & +\infty & & & \\ 0 & +\infty & 2 & & & \end{array} \qquad \mathbf{m}_2 = \ldots$$

5 Domain Operations

In this section, we describe the operations of the domain based on DBM and MMM structures, including transfer functions, reduction, union and widening.

5.1 Transfer Functions

Transfer functions have two purposes:

- compute a sound post-condition for a program statement, that is accounting for all concrete states reachable after executing the statement from a given pre-state;
- report alarms for operations that could not be proved exempt of type error.

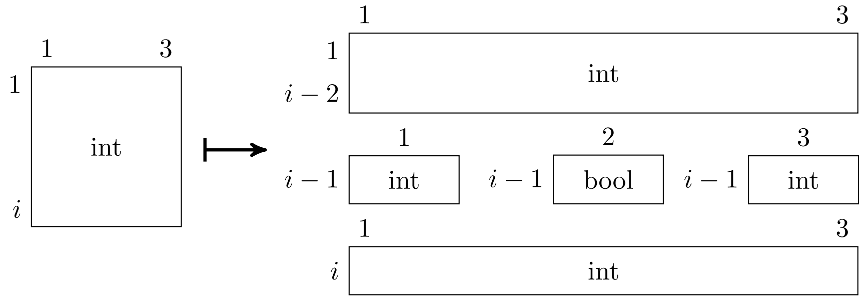
The alarm reporting reduces to the checking that all operations are applied to data of valid type. For instance, if a program statement contains expression $i + \mathbf{Sheet}(k, l)$, and the analysis current abstract state at that point is of the form $(\mathbf{d}, \{(\mathbf{m}_0, t_0), \ldots, (\mathbf{m}_n, t_n)\})$, then it should check that for all $\rho_v \in \gamma_{\mathrm{i}}(\mathbf{d})$ there exists j such that $0 \leq j \leq n$ and $(\overline{x} = k|_{\rho_v}, \overline{y} = l|_{\rho_v}) \in \mathbf{m}_j|_{\rho_v}$, which guarantees that cell $\mathbf{Sheet}(k, l)$ has integer type. In the following, we discuss the computation of post-conditions only.

Assignment Transfer Function. Assignment instructions are either of the form $x := e$ where x is a program variable and e is an expression or of the form $\textbf{Sheet}(e_0, e_1) := e$ where e_0, e_1 and e are expressions. In the first case, the standard assignment transfer function of DBMs shown in [15] leads to define a sound result for transfer function $\textbf{assign}^\sharp$ in the combined domain (when the right hand side e reads a spreadsheet cell, we conservatively assume this read operation may return any possible value, as our abstraction does not carry a precise information about the values stored in the spreadsheet besides type).

In the second case, the typed zones abstractions need to be updated. Let us assume we are computing an abstract post-condition for abstract state $X^\sharp = (\mathbf{d}, \{(\mathbf{m}_0, t_0), \ldots, (\mathbf{m}_n, t_n)\})$. Then:

- when the cell modified in the assignment can be proved to belong to zone $\mathbf{m}_k$ the type of the right hand side is t_k, then typed zone abstractions do not need be modified, and $X^\sharp$ is a valid abstract post-condition;
- otherwise zones need be updated, by removing zone information that may not be preserved in the assignment operation and adding a new zone reduced to the cell that has been modified.

Let us illustrate by an example with an abstract value $X^\sharp = (\mathbf{d}, \{(\mathbf{m}, t)\}) = (\{i : [3, 5]\}, \{(\{\overline{x} : [1, i], \overline{y} : [1, 3]\}, \text{int})\})$. For $\textbf{Sheet}(i-1, 2) := 5$, the assignment transfer function infers that $\gamma_{\mathrm{i}}(\overline{x} = i - 1 \wedge \overline{y} = 2) \subseteq \gamma_{\mathrm{i}}(\mathbf{m})$ under $\mathbf{d} = \{i : [3, 5]\}$, and the type of the expression on the right of the assignment is same as the one of the zone, so $X^\sharp$ remains the same. However, for $\textbf{Sheet}(i-1, 2) :=$ true, the type of the expression of the assignment value is different, if $\textbf{Sheet}(i - 1, 2)$ is within a zone reserved to int, an assignment error Ω_{assign} will be raised, otherwise the abstract zone needs to be split as shown below.



Many zone splitting strategies could be used, e.g. either vertically first or horizontally first. All strategies would yield a sound result. Reduction (Sect. 5.2) tends to remove unnecessary partitions, so that the choice of the splitting strategy is not so crucial.

Condition Test Transfer Function. Condition tests are analyzed with a $\textbf{guard}^\sharp$ abstract function, which inputs an abstract state $X^\sharp$ and a condition c and computes an over-approximation of the concrete states in $\gamma_\times(X^\sharp)$ such that c

evaluates to true. When c involves only program variables, we simply let $\mathbf{guard}^\sharp$ call the condition test function of DBMs [15]. When c involves spreadsheet cells, we let it return $X^\sharp$ (which is always a sound result) as our abstraction ignores spreadsheet values.

5.2 Reduction

As we can see in Sect. 5.1, assignments may generate additional abstract zones, resulting in increasingly large sets of zones. For instance, constraints $(\{\overline{x} : [1, i-1], \overline{y} = 2\}, \text{bool}), \{(\{\overline{x} = i, \overline{y} = 2\}, \text{bool})\}$ could be described by just one constraint. Performing this simplification is the purpose of a (partial) reduction operator, by merging zones when this can be done with no loss in precision.

In the following we let $\mathbf{d}^*$ denote the *closure* of $\mathbf{d}$, the *closure* $\mathbf{m}^*$ *associated with* $\mathbf{d}^*$ is obtained by computing $\mathbf{m}^*$ from the information in both $\mathbf{m}$ and $\mathbf{d}^*$. We write $\vee$ for the point-wise least upper bound over DBMs (resp., MMMs), thus $(\mathbf{d} \vee \mathbf{d}')_{ij} = \max(\mathbf{d}_{ij}, \mathbf{d}'_{ij})$ (resp., $(\mathbf{m} \vee \mathbf{m}')_{ij} = \max(\mathbf{m}_{ij}, \mathbf{m}'_{ij})$). We allow the intersection $\wedge$ over an MMM and a DBM if their sizes and the variables they describe are consistent, the result is a DBM. We define the intersection of a $(h \times l - h' \times l')$ MMM $\mathbf{m}$ and a $h' \times l'$ DBM $\mathbf{d}$ by:

$$\begin{cases} \mathbf{d}'_{i+h-h',j+l-l'} \overset{\triangle}{=} \mathbf{d}_{ij} & \text{if } (i,j) \in [1 \times h'] \times [1 \times l'] \\ \mathbf{d}'_{ij} \overset{\triangle}{=} \mathbf{m}_{ij} & \text{otherwise} \end{cases}$$

We assume an abstract value $(\mathbf{d}, \{(\mathbf{m}_0, t_0), \ldots, (\mathbf{m}_n, t_n)\})$ is given. Let us first look at a pair of its zones $(\mathbf{m}_i, t_i)$ and $(\mathbf{m}_j, t_j)$. Obviously we don't consider merging the two zones if $t_i \neq t_j$. In the other case, we first carry out the normalization, and obtain the closures $\mathbf{d}^*$, $\mathbf{m}^*_i$ and $\mathbf{m}^*_j$ associated with $\mathbf{d}^*$. Then we let $\mathbf{m}^\vee$ be the result of $\mathbf{m}^*_i \vee \mathbf{m}^*_j$, which ensures that $\mathbf{m}^\vee \wedge \mathbf{d}^*$ is an upper bound for $(\mathbf{m}^*_i \wedge \mathbf{d}^*)$ and $(\mathbf{m}^*_j \wedge \mathbf{d}^*)$. But we consider merging these two zones only when $(\mathbf{m}^\vee \wedge \mathbf{d}^*)$ is an exact join of $(\mathbf{m}^*_i \wedge \mathbf{d}^*)$ and $(\mathbf{m}^*_j \wedge \mathbf{d}^*)$, otherwise the merged zone would be less precise than the two initial zones. To verify if $(\mathbf{m}^\vee \wedge \mathbf{d}^*) = (\mathbf{m}^*_i \wedge \mathbf{d}^*) \vee (\mathbf{m}^*_j \wedge \mathbf{d}^*)$, we use the algorithm "Exact Join Detection for Integer Bounded Difference Shapes" introduced in [16], which consists in finding a 4-tuple (i, j, l, k) such that $w_1(i,j) < w_2(i,j) \wedge w_2(k,l) < w_1(k,l) \wedge w_1(i,j) + w_2(k,l) + 2 \leq w(i,l) + w(k,j)$, where w_k and w represent respectively the difference matrices of the 2 operands and the result of the join. If such a 4-tuple exists, the join is not exact. Overall, the reduction algorithm attempts to merge pairs of zone constraints with equal type. Then, the merging rule of two abstract typed zones writes down:

$$\{(\mathbf{m}_i, t_i), (\mathbf{m}_j, t_j)\} \xrightarrow{\substack{t_i = t_j \text{ and} \\ (\mathbf{m}^\vee \wedge \mathbf{d}^*) \text{ is an exact join of } (\mathbf{m}^*_i \wedge \mathbf{d}^*) \text{ and } (\mathbf{m}^*_j \wedge \mathbf{d}^*)}} \{(\mathbf{m}^\vee, t_i)\}$$

where $\mathbf{m}^\vee \overset{\triangle}{=} \mathbf{m}_i \vee \mathbf{m}_j$. In the above example, the following reduction can be performed:

$$\begin{aligned} &(\{i : [1, +\infty]\}, \{(\{\overline{x} : [1, i-1], \overline{y} = 2\}, \text{bool}); \{(\{x = i, y = 2\}, \text{bool})\}\}) \\ \longrightarrow\ &(\{i : [1, +\infty]\}, \{(\{\overline{x} : [1, i], \overline{y} = 2\}, \text{bool})\}) \end{aligned}$$

Now given the whole abstract value $(\mathbf{d}, \{(\mathbf{m}_0, t_0), \ldots, (\mathbf{m}_n, t_n)\})$ which may contain several typed zones, we compute the normalization of all the zones at once. Then reduction picks one zone $\mathcal{Z}_i$, and goes through the other zones, looks for a zone that can be merged with $\mathcal{Z}_i$. A join needs to be calculated, and if the join is exact, the reduction merges both zones into one new zone and proceeds with the other zones. The complexity of a normalization (Floyd-Warshall algorithm) is $O(l^3)$, where l is the length of the side of $\mathbf{m}_k$ (number of program and index variables). The most costly part of the algorithm, the exact join detection, has a worst-case complexity bound in $O(l^3 + r_1 r_2)$, where r_k is the number of edges in difference matrix w_r, but the detection may finish quickly when the join is not exact, which occurs often in practice. Overall the worst-case complexity of the reduction is $O(n^2 \times l^3)$. The algorithm is sound:

Theorem 1 (Soundness). *$r^\sharp$ is an abstract reduction,*

$$\gamma_\times(X^\sharp) \subseteq \gamma_\times(r^\sharp(X^\sharp))$$

5.3 Upper Bound Operator

We now propose an algorithm to compute upper bounds for typed zone constraints, which will also serve as a basis for a widening operator (Sect. 5.4).

We assume two abstract values $X_0^\sharp, X_1^\sharp \in \mathbb{D}_\times^\sharp$ are given, and we assume $X_k^\sharp = (\mathbf{d}_k, \{(\mathbf{m}_k, t)\})$ (the cases where types do not match or where there are several zones will be discussed afterward). Property 3 and Property 4 (Sect. 4.1) provide a straightforward way to compute a common over-approximation for $X_0^\sharp$ and $X_1^\sharp$: indeed, if we let $\mathbf{d} = \mathbf{d}_0 \vee \mathbf{d}_1$ and $\mathbf{m} = \mathbf{m}_0 \wedge \mathbf{m}_1$, we clearly have $\gamma_\times(X_i^\sharp) \subseteq \gamma_\times(\mathbf{d}, \{\mathbf{m}, t\})$, thus $X^\sharp = (\mathbf{d}, \{\mathbf{m}, t\})$ provides a sound upper bound.

Unfortunately, this straightforward technique is not very precise. Indeed, let us consider the case of $X_0^\sharp = (\{i : [2, 2]\}, \{(\{\overline{x} : [i+6, i+6], \overline{y} = 1\}, \mathrm{int})\})$ and $X_1^\sharp = (\{i : [3, 3]\}, \{(\{\overline{x} : [i+5, i+6], \overline{y} = 1\}, \mathrm{int})\})$ (these abstract elements would naturally arise in the first two iterations over a loop that fills a zone with integer values). Then, we obtain $\mathbf{d} = \{i : [2, 3]\}$ and $\mathbf{m} = \{\overline{x} : [i+6, i+6], \overline{y} = 1\}$. While the variable abstraction is satisfactory, the zone is not precise: both $X_0^\sharp$ and $X_1^\sharp$ express the existence of a zone of values of type int with bounds $8 \leq \overline{x} \leq i + 6 \wedge \overline{y} = 1$. When $i = 3$, that zone contains two cells, whereas the zone described by $\{\overline{x} : [i+6, i+6], \overline{y} = 1\}$ only remembers that $\mathbf{Sheet}(9, 1)$ has type int, but forgets about $\mathbf{Sheet}(8, 1)$.

In order to avoid such imprecision, the abstract join algorithm should perform some rewriting steps on both inputs before it actually computes the lower bound on zones. In the case of our example, $X_0^\sharp$ is equivalent (in the sense of $\gamma_\times$) to $(\{i : [2, 2]\}, \{(\{\overline{x} : [8, i+6], \overline{y} = 1\}, \mathrm{int})\})$ and $X_1^\sharp$ is equivalent to $(\{i : [3, 3]\}, \{(\{\overline{x} : [8, i+6], \overline{y} = 1\}, \mathrm{int})\})$. Applying the lower bounds on zone constraints will then produce the desired result. These transformations do not modify the concretization, as shown by Property 2.

For a better illustration, we summarize the general algorithm in Figure 4 and show a step-by-step execution of the algorithm, on the case of the above example

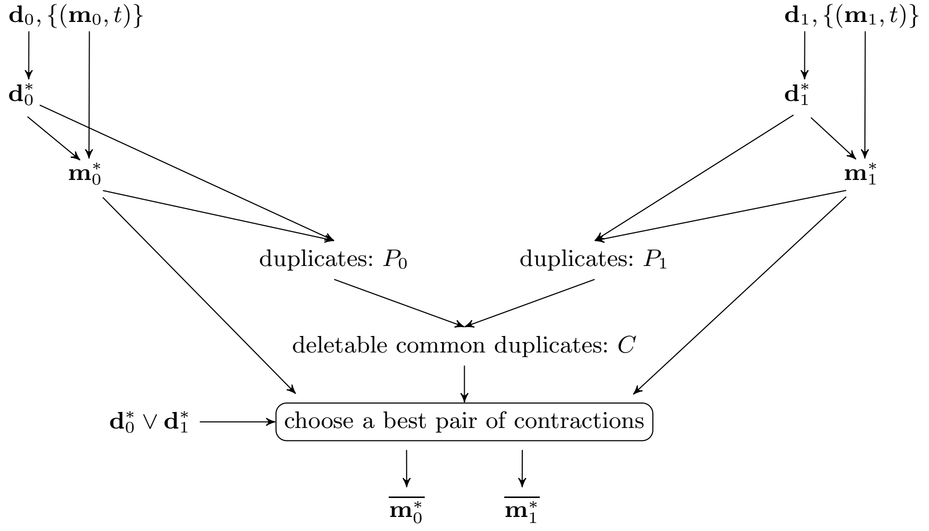

$\mathbf{d}_0, \{(\mathbf{m}_0, t)\}$
$\mathbf{d}_1, \{(\mathbf{m}_1, t)\}$
$\mathbf{d}_0^*$
$\mathbf{d}_1^*$
$\mathbf{m}_0^*$
$\mathbf{m}_1^*$
duplicates: P_0
duplicates: P_1
deletable common duplicates: C
$\mathbf{d}_0^* \vee \mathbf{d}_1^*$
choose a best pair of contractions
$\overline{\mathbf{m}_0^*}$
$\overline{\mathbf{m}_1^*}$

Fig. 4: $(\mathbf{d}_0, \{(\mathbf{m}_0, t)\}) \sqcup^\sharp (\mathbf{d}_1, \{(\mathbf{m}_1, t)\}) \triangleq (\mathbf{d}_0^* \vee \mathbf{d}_1^*, \{(\overline{\mathbf{m}_0^*} \wedge \overline{\mathbf{m}_1^*}, t)\})$

in Figure 5. For the sake of simplicity, we omit the constraint $\overline{y} = 1$ as it is appears in both operands and can be handled trivially. So the general algorithm consists of five steps:

– *Normalize.* Given two abstract values $(\mathbf{d}_k, \{(\mathbf{m}_k, t)\})$, we first carry out the normalization, and obtain the closures $\mathbf{d}_k^*$ and $\mathbf{m}_k^*$ associated with $\mathbf{d}_k^*$.

– *Calculate Duplicates.* A difference matrix can be seen as a representation of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A}, w)$ with weighted edges. From its (shortest-path) closure, there are some edges which can be *restored* by other edges. E.g., in $\mathbf{m}_0^* \wedge \mathbf{d}_0^*$, $w(\overline{x}, i) = -6$ is *deletable*, because it can be restored by the sum of $w(\overline{x}, 0) = -8$ and $w(0, i) = 2$. We say $w(\overline{x}, i)$ is a *duplicate* of $w(\overline{x}, 0)$ and $w(0, i)$, and we let $(\overline{x}, i) \hookleftarrow 0$ denote this duplication. A *contraction* $\overline{\mathbf{m}^*}$ of an MMM $\mathbf{m}^*$ *associated with* $\mathbf{d}^*$, refers to an MMM deleting some duplicates, and $\mathbf{m}^*$ can still be restored from $\overline{\mathbf{m}^*} \wedge \mathbf{d}^*$. E.g., $\mathbf{m}_0^* \wedge \mathbf{d}_0^*$ without $w(\overline{x}, i)$ is actually one contraction of $\mathbf{m}_0^* \wedge \mathbf{d}_0^*$. So this step aims at finding the set of all the possible duplicates P_k in $\mathbf{m}_k^* \wedge \mathbf{d}_k^*$.

– *Find Deletable Common Duplicates.* Considering now the 2 operands together, we can find the set of the common duplicates of the 2 operands: $P_0 \cap P_1$. Then some subsets of this set, which are actually some common duplicates, can be deleted from both operands to compute two contractions. This step searches for the set of this kind of subsets that C denotes.

– *Choose a Best Pair of Contractions.* Taking the 2 operands of our example, C contains both $\{(\overline{x}, i), (0, \overline{x})\}$ and $\{(i, \overline{x}), (\overline{x}, 0)\}$. Although the concretization of a contraction is always the same as the one of the original matrix, if we forecast the next step – the intersection of both contractions, the choice of the

Initial operands:

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -2 \\ 0 & 2 & 0 \end{array} \quad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & -6 & \\ \overline{y} & & 0 & & \\ \cline{4-5} i & 6 & & & \\ 0 & & & & \end{array}\right), \mathrm{int} \qquad \begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -3 \\ 0 & 3 & 0 \end{array} \quad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & -5 & \\ \overline{y} & & 0 & & \\ \cline{4-5} i & 6 & & & \\ 0 & & & & \end{array}\right), \mathrm{int}$$

After normalization:

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -2 \\ 0 & 2 & 0 \end{array} \quad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & -6 & -8 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & 6 & & 0 & -2 \\ 0 & 8 & & 2 & 0 \end{array}\right), \mathrm{int} \qquad \begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -3 \\ 0 & 3 & 0 \end{array} \quad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & -5 & -8 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & 6 & & 0 & -3 \\ 0 & 9 & & 3 & 0 \end{array}\right), \mathrm{int}$$

2 sets of duplicates:

$P_0 = \{(\overline{x}, i) \hookleftarrow 0, (\overline{x}, 0) \hookleftarrow i,$
$(i, \overline{x}) \hookleftarrow 0, (0, \overline{x}) \hookleftarrow i, \ldots\}$

$P_1 = \{(\overline{x}, i) \hookleftarrow 0, (\overline{x}, 0) \hookleftarrow i,$
$(i, \overline{x}) \hookleftarrow 0, (0, \overline{x}) \hookleftarrow i, \ldots\}$

Sets of deletable common duplicates:

$$C = \{\{(\overline{x}, i), (0, \overline{x})\}, \{(\overline{x}, i), (i, \overline{x})\}, \{(\overline{x}, 0), (0, \overline{x})\}, \{(i, \overline{x}), (\overline{x}, 0)\}, \ldots\}$$

Choose a best pair of contractions:

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -2 \\ 0 & 2 & 0 \end{array} \quad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & & -8 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & 6 & & 0 & -2 \\ 0 & & & 2 & 0 \end{array}\right), \mathrm{int} \qquad \begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -3 \\ 0 & 3 & 0 \end{array} \quad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & & -8 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & 6 & & 0 & -3 \\ 0 & & & 3 & 0 \end{array}\right), \mathrm{int}$$

Final join:

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -2 \\ 0 & 3 & 0 \end{array} \quad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & & -8 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & 6 & & & \\ 0 & & & & \end{array}\right), \mathrm{int}$$

Fig. 5: Computation of a join

set of duplicates to delete, resulting in the contractions, makes actually a real difference: Both contractions formed by deleting $\{(\overline{x}, i), (0, \overline{x})\}$ gives a larger intersection than the ones formed by deleting $\{(i, \overline{x}), (\overline{x}, 0)\}$. So based on C, $\mathbf{m}_k^*$ and $\mathbf{d}_0^* \vee \mathbf{d}_1^*$, this step finds a set of duplicates to delete, thus computes a pair of contractions $\overline{\mathbf{m}_k^*}$ for the next step.

– *Join of DBMs and Intersection of MMMs.* The final step joins the two transformed operands by joining their DBMs and intersecting their MMMs.

As all steps either preserve concretization or return an over-approximation of the arguments (under-approximating zones), this algorithm is sound:

Theorem 2 (Soundness). *With the above notations:*

$$\gamma_\times(X_0^\sharp) \cup \gamma_\times(X_1^\sharp) \subseteq \gamma_\times(X^\sharp)$$

After normalization:

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -1 \\ 0 & 1 & 0 \end{array} \qquad (\bot_{\mathbf{zone}}, \mathrm{int})$$

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -2 \\ 0 & 2 & 0 \end{array} \qquad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & 1 & -1 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & -1 & & \multicolumn{1}{|c}{0} & -2 \\ 0 & 1 & & \multicolumn{1}{|c}{2} & 0 \end{array}, \mathrm{int} \right)$$

Sets of duplicates:

$P_0 =$ not applicable

$$P_1 = \{(\overline{x}, i) \hookleftarrow 0, (\overline{x}, 0) \hookleftarrow i, (i, \overline{x}) \hookleftarrow 0, (0, \overline{x}) \hookleftarrow i, \ldots\}$$

Sets of deletable duplicates:

$D_0 =$ not applicable

$$D_1 = \{\{(\overline{x}, i), (0, \overline{x})\}, \{(\overline{x}, i), (i, \overline{x})\}, \{(\overline{x}, 0), (0, \overline{x})\}, \{(i, \overline{x}), (\overline{x}, 0)\}, \ldots\}$$

Choose a best contraction:

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -1 \\ 0 & 1 & 0 \end{array} \qquad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & & -1 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & -1 & & \multicolumn{1}{|c}{0} & -1 \\ 0 & & & \multicolumn{1}{|c}{1} & 0 \end{array}, \mathrm{int} \right)$$

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -2 \\ 0 & 2 & 0 \end{array} \qquad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & & -1 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & -1 & & \multicolumn{1}{|c}{0} & -2 \\ 0 & & & \multicolumn{1}{|c}{2} & 0 \end{array}, \mathrm{int} \right)$$

Final join:

$$\begin{array}{c|cc} & i & 0 \\ \hline i & 0 & -1 \\ 0 & 2 & 0 \end{array} \qquad \left(\begin{array}{c|cccc} & \overline{x} & \overline{y} & i & 0 \\ \hline \overline{x} & 0 & & & -1 \\ \overline{y} & & 0 & & \\ \cline{4-5} i & -1 & & \multicolumn{1}{|c}{} & \\ 0 & & & \multicolumn{1}{|c}{} & \end{array}, \mathrm{int} \right)$$

Fig. 6: Creation of a new typed zone in join

Computation of a New Zone. So far, we focused on the case where both operands of $\sqcup^\sharp$ consist of exactly one zone. In practice, most cases fall out of this scope. We consider here the case where the left argument contains no zone and the right operand contains one zone, and we will treat the general case in the next paragraph. This case is typically encountered when computing an abstract join after the first iteration of a loop that initializes a spreadsheet zone. For instance, such a program would give us abstract states $X_0^\sharp = (\{i = 1\}, \emptyset)$ at iteration 0 and $X_1^\sharp = (\{i = 2\}, \{(\{\overline{x} = i - 1, \overline{y} = 2\}, \mathrm{int})\})$ at iteration 1. Then $X_0^\sharp$ is actually equivalent to abstract state $(\{i = 1\}, \{(\bot_{\mathbf{zone}}, \mathrm{int})\})$ where $\bot_{\mathbf{zone}}$ denotes the MMM with empty concretization, hence the empty zone. We remark that the constraints of the zone can in both cases be rewritten into $\{1 \leq \overline{x} \leq i - 1, \overline{y} = 2\}$: indeed, when $i = 1$, this is equivalent to the empty MMM. Thus, $(\{i : [1, 2]\}, \{(\{\overline{x} : [1, i - 1], \overline{y} = 2\}, \mathrm{int})\})$ is an over-approximation for both operands, hence a valid result for $\sqcup^\sharp$.

We assume that operands are of the form $X_0^\sharp = (\mathbf{d}_0, \{(\bot_{\mathbf{zone}}, t)\})$ and $X_1^\sharp = (\mathbf{d}_1, \{(\mathbf{m}_1, t)\})$. Then, we follow the algorithm given in the case of two abstract states with exactly one zone up to the step normalization. Then for the following two steps about duplicates, as the zone of the left operand is $\bot_{\mathbf{zone}}$, we calculate only the part of the right operand. Then for the step of choosing a contraction, we search for a set of deletable duplicates in D_1 to delete, thus computes a contraction $\overline{\mathbf{m}_1^*}$ of $\mathbf{m}_1^*$ associated with $\mathbf{d}_1^*$ (therefore $\gamma_{\mathrm{i}}(\overline{\mathbf{m}_1^*} \wedge \mathbf{d}_1^*) = \gamma_{\mathrm{i}}(\mathbf{m}_1^* \wedge \mathbf{d}_1^*)$) , such that $\overline{\mathbf{m}_1^*} \wedge \mathbf{d}_0 = \emptyset$. If such a $\overline{\mathbf{m}_1^*}$ can be found, $\sqcup^\sharp$ keeps it for MMMs of both operands, and computes a join for the DBMs of both operands. Otherwise, the right hand zone is discarded. Fig. 6 shows this algorithm on the above example.

Case of an Arbitrary Number of Zones. We now consider the case of two $\mathbb{D}^\sharp_\times$ elements $X^\sharp_k = (\mathbf{d}_k, \{(\mathbf{m}_{k,0}, t_{k,0}), \ldots, (\mathbf{m}_{k,n}, t_{k,n})\}), k \in \{0, 1\}$. In that case, abstract join operator $\sqcup^\sharp$ should identify pairs of zones that can be over-approximated with minimal loss of precision, and zones in the right hand side argument that can be joined with an empty zone with no loss of precision. Precisely, the steps of the normalization and the calculation of duplicates can be first done on every zone of both operands at once. Then for one zone $(\mathbf{m}_{0,i}, t_{0,i})$ of $X^\sharp_0$, the algorithm goes through the zones of $X^\sharp_1$, proceed the step of deletable common duplicates and see if an optimal pair of contractions can be found. If so, the algorithm considers the pair of zones is identified; otherwise, it continues to examine the rest of the zones in $X^\sharp_1$. In the end if a pair of zones is identified, it adds their join to the result set, and remove them from both operands; otherwise, it adds the join of $(\mathbf{m}_{0,i}, t_{0,i})$ and $(\bot_{\mathbf{zone}}, t_{0,i})$ to the result set, and remove the zone from $X^\sharp_0$. The whole algorithm proceeds this way for each zone in $X^\sharp_0$. The most costly step of a join of 2 zones is to compute the sets of deletable common duplicates C: larger the sets of duplicates and their intersection are, more computation it requires. The complexity of the step to choose a best pair of contractions is proportional to the size of C and the one of each element. Finally the number of zones in each operand and the size of $\mathbf{m}_k$ also determines the complexity of the entire operation.

5.4 Widening Operator

Abstract join operator $\sqcup^\sharp$ shown in Sect. 5.3 returns an upper bound of its argument, but does not enforce termination of abstract iterates. However, we can extend $\sqcup^\sharp$ into a widening operator as follows:

- we let $\nabla^\sharp$ use a widening operator $\nabla^\sharp_{\mathbf{d}}$ over DBMs instead of $\vee$;
- after a fixed number of iterations $N_{\nabla^\sharp}$, the steps of the computation of $\overline{\mathbf{m}_k^*}$ and their intersection are replaced by a lower bound computation:

$$\overline{\mathbf{m}_0^*} \wedge \overline{\mathbf{m}_1^*} = \begin{cases} \mathbf{m}_0^* & \text{if } \forall i, j, (\mathbf{m}_0^*)_{ij} \leq (\mathbf{m}_1^*)_{ij}, \\ \bot_{\mathbf{zone}} & \text{otherwise} \end{cases}$$

  (in practice, empty zones are pruned out of $\mathbb{D}^\sharp_{\mathbf{Z}}$ elements).

This provides a sound and terminating widening operator $\nabla^\sharp$ over $\mathbb{D}^\sharp_\times$.

5.5 Analysis

Transfer functions shown in Sect. 5.1 and the reduction, join and widening operators of Sect. 5.2-5.4 allow us to define a standard abstract interpretation based static analysis for the restricted spreadsheet language of Sect. 3. Our analysis implements a classic iteration engine over the program control flow graphs, and performs widening at loop heads. We use a delayed widening iteration strategy, where the regular join operator $\sqcup^\sharp$ is used in the first iterations over each loop, and $\nabla^\sharp$ is used for the following iterations. The reduction operator of Sect. 5.2 is used after the computation of transfer functions which modify the structure of zones. It is not applied to the widening output, as this might break termination. Our analysis is sound in the sense of the correctness theorem below:

Theorem 3 (Correctness).
If (l, ρ) is reachable for $\rightarrow$, then $\rho \in \gamma_\times(X_l^\sharp)$ where $X_l^\sharp$ is the invariant at l.
If (l, ρ) is reachable for $\rightarrow$, and $(l, \rho) \rightarrow \Omega$, then an alarm is reported at l.

6 Prototype and Results

The analysis was implemented in OCaml and represents around 3000 lines of code, including a front-end for our restricted spreadsheet language. We have applied our analysis to a number of small programs and examined type properties of the arrays that they manipulate. We ran the analysis on programs consisting of a single loop as well as programs with nested loops. In the table, we show the size in pre-processed lines of code and the analysis time without any spurious type warning on a 2.80 GHz Intel Core Duo with 4GB RAM. The analyzer raises various type errors (e.g., Ω_{assign}) if they exist in the programs.

Benchmark	Loop Level	Code Size (loc)	Run Time (sec)
initialization of a row	1	13	0.042
creation of 2 columns (program 1)	1	31	0.258
copy of a matrix	2	20	0.071
insertion sort of a column	2	29	0.135
multiplication of 2 matrices	3	35	0.096

7 Conclusion and Future Work

Our proposal enables static analysis of spreadsheet programs and verifies that an important class of type errors will not occur. It is based on a combination of numeric abstraction to describe spreadsheet zones and a type abstraction over cell contents.

The upper bound operators of our abstract domain accommodates an under-approximation operation of a sub-domain. [17] presents generic procedures that work for any base domain to compute under-approximations. In comparison with their approach, our domain is adapted specifically for the application, thus closer to a precise and efficient analysis for spreadsheet programs.

Substituting other lattices to the type lattice used in this paper will allow us to carry out other analyses. E.g. in practice we may relax the exact type characterization and permit approximate types (e.g. “int or bool”) to more compactly capture zones which maybe otherwise need to be split to a large number of smaller zones. Existing work has considered other type properties, such as units and dimensions properties [18,19] (e.g., distinguishing hours, minutes, seconds, etc.), albeit only at the interface level, whereas we are considering the spreadsheet programs. Our work could be extended to deal with notions of units in a very straightforward manner by only substituting lattices. Information to build that lattice could be determined from header and labels in the spreadsheets.

Another important extension of our work would be to deal with a full spreadsheet language instead of the restricted language considered in this paper, so as to analyze industrial applications.

Our work also opens some more theoretical abstract domain design issues. In particular, it would be interesting to explore other instantiations of the abstract domain, with other kinds of numerical constraints over zones. For instance, we may consider octagons [20] (also allowing constraints of the form $\overline{x} + i \geq c$ where i is a program variable), or simple disequalities [21]. This would require a more general representation of zone constraints, and operators to cope with this more general representation. Last, it would also be interesting to extend array content analysis such as [22,23] to our two dimensional zones, so as to discover relations between program variable data and more complex properties of contents of spreadsheet zones.

Acknowledgments. We would like to thank Antoine Miné, Enea Zaffanella and members of the EuSpRIG (European Spreadsheet Risks Interest Group) for helpful discussions. We are grateful to the referees for their encouraging and useful comments on the early version of the article.

References

1. Panko, R.R.: What we know about spreadsheet errors. Journal of End User Computing **10** (1998) 15–21
2. Jones, S.P., Blackwell, A., Burnett, M.: A user-centred approach to functions in excel. In: ICFP ’03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, ACM (2003) 165–176
3. Sestoft, P.: Implementing function spreadsheets. In: WEUSE ’08: Proceedings of the 4th international workshop on End-user software engineering, New York, NY, USA, ACM (2008) 91–94
4. Cheng, T.: Excel Functional Programming. Explore another dimension of spreadsheet programming (2010)
5. Wakeling, D.: Spreadsheet functional programming. J. Funct. Program. **17**(1) (2007) 131–143
6. Erwig, M., Abraham, R., Kollmansberger, S., Cooperstein, I.: Gencel: a program generator for correct spreadsheets. J. Funct. Program. **16** (May 2006) 293–325

7. Abraham, R., Erwig, M.: Inferring templates from spreadsheets. In: Proceedings of the 28th international conference on Software engineering. ICSE '06, New York, NY, USA, ACM (2006) 182–191
8. Silva, A.: Strong Types for Relational Data Stored in Databases or Spreadsheets. PhD thesis, University of Minho (2006)
9. Cunha, J., Saraiva, J., Visser, J.: From spreadsheets to relational databases and back. In: PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, ACM (2009) 179–188
10. Rajalingham, K., Chadwick, D.R., Knight, B.: Classification of spreadsheet errors. In: EuSpRIG 2000 Symposium: Spreadsheet Risks, Audit and Development Methods. (2001)
11. Bradley, L., McDaid, K.: Using bayesian statistical methods to determine the level of error in large spreadsheets. In: ICSE Companion. (2009) 351–354
12. Bishop, B., McDaid, K.: Spreadsheet debugging behaviour of expert and novice end-users. In: Proceedings of the 4th international workshop on End-user software engineering. WEUSE '08, New York, NY, USA, ACM (2008) 56–60
13. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, ACM Press, New York, NY (1979) 269–282
14. Cheng, T.: Verification of spreadsheet programs by abstract interpretation. Master's thesis, École Polytechnique (2011)
15. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Proc. of the 2d Symp. on Programs as Data Objects (PADO II). Volume 2053 of Lecture Notes in Computer Science., Springer (May 2001) 155–172
16. Bagnara, R., Hill, P.M., Zaffanella, E.: Exact join detection for convex polyhedra and other numerical abstractions. Computational Geometry: Theory and Applications **43**(5) (2010) 453–473
17. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '08, New York, NY, USA, ACM (2008) 235–246
18. Chambers, C., Erwig, M.: Automatic detection of dimension errors in spreadsheets. Journal of Visual Languages and Computing **20**(4) (2009) 269–283
19. Antoniu, T., Steckler, P.A., Krishnamurthi, S., Neuwirth, E., Felleisen, M.: Validating the unit correctness of spreadsheet programs. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 439–448
20. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1) (2006) 31–100
21. Péron, M., Halbwachs, N.: An abstract domain extending difference-bound matrices with disequality constraints. In: VMCAI'07: Eighth International Conference on Verification, Model Checking, and Abstract Interpretation. Volume 4349 of Lecture Notes in Computer Science., Springer (January 2007) 268–282
22. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: PLDI'08: 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM (June 2008) 339–348
23. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Conference Record of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY (2011) 105–118