JFP **30**, e26, 68 pages, 2020. © Microsoft 2020
doi:10.1017/S0956796820000234

Elastic sheet-defined functions: Generalising spreadsheet functions to variable-size input arrays[*]

MATT MCCUTCHEN[1]
Massachusetts Institute of Technology
(*e-mail:* matt@mattmccutchen.net)

JUDITH BORGHOUTS[1]
University of California Irvine
(*e-mail:* jborghou@uci.edu)

ANDREW D. GORDON
Microsoft Research and University of Edinburgh
(*e-mail:* adg@microsoft.com)

SIMON PEYTON JONES
Microsoft Research
(*e-mail:* simonpj@microsoft.com)

ADVAIT SARKAR
Microsoft Research and University of Cambridge
(*e-mail:* advait@microsoft.com)

Abstract

Sheet-defined functions (SDFs) bring modularity and abstraction to the world of spreadsheets. Alas, end users naturally write SDFs that work over *fixed-size* arrays, which limits their re-usability. To help end user programmers write more re-usable SDFs, we describe a principled approach to *generalising* such functions to become *elastic SDFs* that work over inputs of arbitrary size. We prove that under natural, checkable conditions our algorithm returns the principal generalisation of an input SDF. We describe a formal semantics and several efficient implementation strategies for elastic SDFs. A user study with spreadsheet users compares the human experience of programming with elastic SDFs to the alternative of relying on array processing combinators. Our user study finds that the cognitive load of elastic SDFs is lower than for SDFs with map/reduce array combinators, the closest alternative solution.

[*] This article has been published in a revised form in the Journal of Functional Programming. This is the authors' version as of July 8, 2020, reformatted on September 21, 2020, and is licensed under CC-BY-NC-ND 4.0. No commercial re-distribution or re-use allowed. Derivative works cannot be distributed. © 2020 Microsoft.
[1] Most of the contribution of these authors was made while they were interning at Microsoft Research.

1 Introduction

Suppose we want to use a spreadsheet to compute the average of the numeric cells[2] in A1:A10. We might do so like this, using a simple textual notation for the spreadsheet grid, described in Section 3:

```
B1 = SUM( A1:A10 ); B2 = COUNT( A1:A10 ); B3 = B1/B2
```

Rather than repeat this logic in many places, it would be better to *define* it once, and *call* it in many places. With this in mind, Peyton Jones et al. (2003) propose that a function can be defined by an ordinary worksheet with specially-identified input and output cells. These *sheet-defined functions* (SDFs) are explored and implemented by Sestoft (2014), in his comprehensive monograph on spreadsheet implementation technology. In our textual notation we might write

```
function AVG( A1:A10 ) returns B3 {
  B1 = SUM( A1:A10 ); B2 = COUNT( A1:A10 ); B3 = B1/B2 }
```

But there is a problem: this function only works on vectors of length 10, unlike the built-in AVG which works on vectors of arbitrary size. We would like to *generalise* AVG, by replacing the fixed 10 by a *length variable* α, thus making an *elastic* SDF:[3]

```
function AVG( A1:A{α} ) returns B3 {
  B1 = SUM( A1:A{α} ); B2 = COUNT( A1:A{α} ); B3 = B1/B2 }
```

This simple example is representative of a large class of problems that a user might want to solve by first defining a function that manipulates individual elements of fixed-size input arrays (because that is easy in spreadsheets), and then somehow generalising the function to work on inputs of arbitrary size. *The core contribution of this paper is an algorithm to perform this generalisation step*. Our specific contributions are these:

- Sarkar et al. (2018) propose a simple textual notation for spreadsheet programs (Section 3). We develop this idea further by introducing new notation for *sheet-defined functions* (SDFs).
- We generalise the SDF notation so that it can describe *elastic SDFs*, which work on inputs of varying size (Section 4), and give our new notation a precise semantics (Section 5) using what we call *tiles*. We also discuss practical execution mechanisms in Section 9.
- Our goal is to find the elastic SDF that is the unique *principal generalisation* of the original SDF. We begin by specifying what it means for one generalisation to be "more general than" another, and identifying a series of obstacles to the very existence of a principal generalisation (Section 6).
- In Section 7 we show that every (suitable) SDF does indeed enjoy a principal generalisation under certain restrictions. Better still, we give an algorithm that finds the principal generalisation, and prove the algorithm correct. However, if the SDF does not use any computational pattern our system can generalise, the principal generalisation may just be the original SDF. In Section 8, we extend the system to support

[2] I.e., ignoring cells that are blank or that contain strings or booleans. Note that SUM and COUNT both do this. We temporarily ignore the fact that many spreadsheet tools have a built-in AVERAGE function that solves the original problem.

[3] We explain in Section 6.4 why we do not also generalise the width of 1 to a variable.

more computational patterns and thereby provide useful generalisations for more SDFs.

- The ultimate goal of programming language research is to make human beings more productive. So in Section 10 we describe a user study in which we asked 20 participants to write array-processing SDFs, either using elastic SDFs or by the method of storing input and output arrays in single cells (the idea of arrays-in-cells (Blackwell et al., 2004)). A key finding is that users perceived a significantly lower cognitive workload for elastic SDFs, versus SDFs with arrays-in-cells.

Spreadsheets have a vastly larger user base than mainstream programming languages, but are seldom studied by programming language researchers. This paper is one of only a handful to apply the formal arsenal of the programming language community to spreadsheets, by giving a textual notation for spreadsheets, a formal semantics, a generalisation ordering, and a principal generalisation theorem. The technical details of elastic SDFs are subtle, but the task is hard: generalising a single, concrete program to a size-polymorphic one, not just with heuristics, but in a provably most-general way. From the point of view of the user, however, things are very simple: you can build a spreadsheet using familiar element-wise formulas and copy/paste, capture that computation as a reusable function (exemplified on inputs of fixed size), and have it reliably and automatically generalised to work on inputs of arbitrary size. That is a valuable prize.

Nothing in this paper is vendor-specific; our generalisation technique relies only on the structure of the spreadsheet grid, and the copy/paste behaviour of absolute and relative references, features that are present in essentially all spreadsheets, and that we describe in Section 3.1.

We break this lengthy paper into sections as follows:

- Section 2 reviews the idea of sheet-defined functions.
- Section 3 introduces a formal textual notation for spreadsheets, used in our technical development.
- Section 4 defines the syntax and informal semantics of elastic SDFs.
- Section 5 describes a formal semantics for SDFs and elastic SDFs.
- Section 6 studies the idea of generalizing from an example SDF to an elastic SDF and the restrictions we need to impose so that a principal (most general) elastic SDF exists.
- Section 7 describes our generalization algorithm and proves it correct.
- Section 8 extends the system of Sections 6 and 7 to provide useful generalisations for more SDFs.
- Section 9 describes several techniques for executing elastic SDFs.
- Section 10 reports our user study.
- Section 11 discusses related work.
- Section 12 draws conclusions.
- Appendix A describes details of a collision-avoidance algorithm used in Section 9.

2 Sheet-defined functions

Fifteen years ago Peyton Jones *et al.* argued that it should be possible for spreadsheet users to define new spreadsheet functions whose implementation is given *by a spreadsheet* – that is, a sheet-defined function (SDF) (Peyton Jones et al., 2003). This approach contrasts with that taken by typical spreadsheet products today: users may define *custom functions* but only by using a conventional programming language separate from the spreadsheet formula language. Microsoft Excel allows custom functions to be defined using Visual Basic or JavaScript, while Google Sheets allows custom functions written in JavaScript.

Requiring a switch of programming language – and indeed of programming *paradigm* – puts a huge barrier between the domain expert using a spreadsheet and their goal of composing existing functions together to build a new one. No other programming language demands that users write functions in a different language – why should spreadsheets be different?

2.1 Is there any demand for SDFs?

The key goal of sheet-defined functions is to *support re-use*, allowing spreadsheet users to write code once, rather than repeatedly. There is clearly very strong demand for such re-use in spreadsheet applications:

- Excel has long allowed users to write custom functions in a dialect of Visual Basic known as VBA (Visual Basic for Applications), an implementation of the programming language Visual Basic 6. The VBA scripting feature is immensely popular, with the query “Excel VBA” currently retrieving over 119,000 results on StackOverflow.[4][5] For context: at present, that is greater than the number of results for 6 out of the 20 most popular programming languages (as ranked by the January 2020 Tiobe index[6]).
- Besides writing VBA scripts, users commonly extend the function library available in Excel by installing add-ins. For example, one add-in[7] that introduces a number of functions to Excel is extremely popular, with over 750,000 users.

These user behaviours (writing VBA and installing custom add-ins) require significant programming expertise and attention investment and, in the case of add-ins, usually a financial investment. The fact that, despite these obstacles, such a large user base engages in these activities *at all* indicates a significant demand for user-defined functions.

Sheet-defined functions require no additional expertise beyond understanding of the formula language, and thus make it possible for a much larger class of end users to build their own abstractions without the assistance of a professional programmer. It is not unreasonable to expect that the use of SDFs will eclipse the use of conventional custom functions

[4] https://stackoverflow.com/questions/tagged/excel+vba. Last accessed: January 6, 2020

[5] It should be noted that sometimes VBA and add-in functions are used not for the straightforward encapsulation of logic but because they have access to Excel or OS environment features that are outside the spreadsheet computational model. Also, a search for “Excel VBA” could pick up uses of VBA for purposes other than defining a function for use in formulas; for instance, making systematic edits to a spreadsheet.

[6] https://www.tiobe.com/tiobe-index/.

[7] https://www.asap-utilities.com/asap-utilities-excel-tools-tip.php?tip=259&utilities=97&lang=en_us. Last accessed: January 6, 2020

or proprietary add-ins. One avenue for validating this idea, which we have not pursued at present, is to analyse a corpus of spreadsheets to study the degree to which functionality appears to be reused in multiple places, to better understand how much we might expect SDFs to penetrate.

2.2 Design space for SDFs

Sestoft took the idea of sheet-defined functions and developed an open-source implementation (Sestoft and Sørensen, 2013; Sestoft, 2014). However, this and other implementations are only points in a broader design space for SDFs. We have identified a few important parameters of this design space, including:

- *User interface.* Through what user interface does the user define a new function, give it a name, and specify its parameters and result?
- *Function sheets.* Is a worksheet that defines the body of a function special in some way (a "function sheet"); or it is just a regular worksheet? Can multiple SDFs be defined in a single worksheet?
- *Debugging.* What is the appropriate debugging support for SDFs? (Peyton Jones et al. (2003) offers some ideas.)
- *Libraries and distribution.* How can SDFs be collected into a library, versioned, and shared with others?
- *Discoverability.* How can SDFs be made discoverable, so that users actually know they exist?

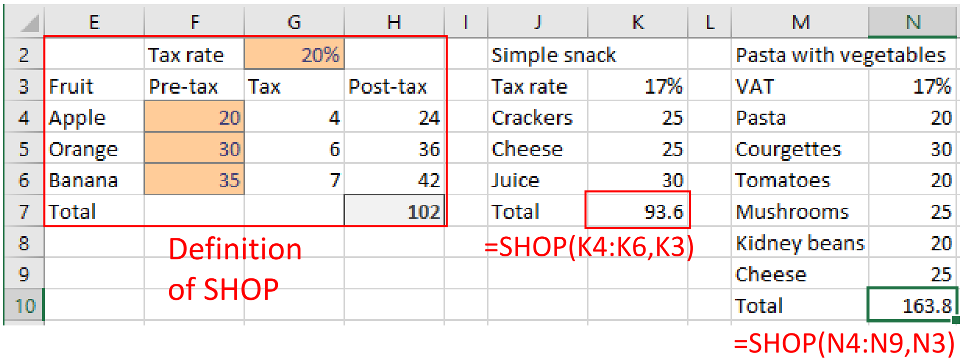
Then, for any particular realisation of SDFs, there are open questions about utility. Will an end user find that SDFs justify their learning costs (Bostrom et al., 1990), and attention investment requirements (Blackwell, 2002)? Do users actually become more productive by using SDFs (Pandita et al., 2018)?

2.3 Elasticity

The questions posed in the previous section are important, but regardless of how these design decisions are resolved, there remains the fundamental challenge of elasticity: *how to build functions that adapt to inputs of varying size*. That (and only that) is the challenge that we address in this paper.

One solution might be to require the user to make full use of arrays as values (Blackwell et al., 2004). In our AVG example, we could say that the entire input array lands in cell A1, and then compute the result as SUM(A1)/COUNT(A1), where the SUM adds up all the elements of its input array, and similarly for COUNT.

The arrays-as-values approach would require the spreadsheet to allow a cell to contain an array value and, in general, to support nested arrays. That might be technically reasonable, but presents a significant practical challenge. Spreadsheet users have decades of experience in doing iterated calculation by copy/paste or drag-fill, working with fixed-size ranges. In our AVG example, it is appealingly concrete to write SUM(A1:A10)/COUNT(A1:A10). Moreover, when using arrays-as-values, many common cases would require the use of



	E	F	G	H	I	J	K	L	M	N
2		Tax rate	20%			Simple snack			Pasta with vegetables	
3	Fruit	Pre-tax	Tax	Post-tax		Tax rate	17%		VAT	17%
4	Apple	20	4	24		Crackers	25		Pasta	20
5	Orange	30	6	36		Cheese	25		Courgettes	30
6	Banana	35	7	42		Juice	30		Tomatoes	20
7	Total			102		Total	93.6		Mushrooms	25
8									Kidney beans	20
9									Cheese	25
10									Total	163.8

Fig. 1: A spreadsheet containing the body of the SHOP SDF and two calls to it, one with a different input size than the original.

maps, zips, folds, and other aggregate operations in the functional programming armory. Even for a functional programmer some of the simple examples in this paper are quite tricky to express, so many users would perhaps never get beyond the fixed-size version.

So the question is this: can we allow users to write SDFs over fixed-size arrays, using familiar element-wise operations and copy/paste; and then reliably and predictably generalise them to work over arbitrary sized arrays? Yes, we can.

3 A textual notation for sheet-defined functions

In this work we deliberately abstract away from the user interface, in order to concentrate on the data and computational aspects of SDFs. But we have to describe SDFs somehow! A simple approach is to describe spreadsheets textually which, as well as slicing away the user interface, allow us to deploy the tools of the programming language community.

We therefore borrow a recently introduced textual notation for describing the data and computations of a spreadsheet grid – the *Calculation View* (Sarkar et al., 2018) of the spreadsheet. The complete Calculation View language as used in this paper is given in Figure 2 below for reference.

We do not expect users to use textual notation to define SDFs: Section 3.5 describes how we expect to get from a familiar spreadsheet grid to a textual SDF. For the purposes of this paper, however, the discussion of generalisation is made much easier by starting from a concrete textual form.

3.1 Range assignments, formulas, and values

Figure 1 shows a spreadsheet that computes, in H7, the total cost of purchasing the items in F4:F6, after accounting for sales tax, whose rate is held in G2. In Calculation View form, the part of the spreadsheet in the red box is written thus (ignoring text labels, which are not involved in the computation):

```
F4 = 20; F5 = 30; F6 = 35; G2 = 20%
G4:G6 = F4 ∗ $G$2
```

```
H4:H6 = F4 + G4
H7    = SUM(H4:H6)
```

Each cell contains a *value* computed by the formula in the cell. A value is a number, boolean, string, error value, or array of values; the exact details are not relevant to this paper.

The Calculation View form of this sheet fragment consists of a set of *range assignments* (note that a range may be a single cell), which can be written in any order, and can appear on successive lines or on a single line separated by semicolons. Each range assignment is of form $t\ r = F$, where t is a *tile name* (explained in Section 3.4 and omitted until then), r is a *range*, and F is a *formula*.

Ranges r are already standard in spreadsheets. Referring to Figure 2, a range r can be denoted by:

- A single *cell address* a, such as H7. In this paper, we use only the standard *A1 notation* Sestoft (2014) where the cell is identified by a column name and a row number.
- A rectangular range denoted by two cell addresses for the upper-left and lower-right corners, such as G4:G6. We call this *corner-corner* notation. Conventional spreadsheets treat a "back to front" range like G5:G4 as identical to G4:G5. This implicit reversal makes it impossible to represent an empty range, something that seems absolutely necessary as we move to size-polymorphic functions. In our work we do no implicit reversal. We consider the range $N_1m_1 : N_2m_2$ to have height $m_2 - m_1 + 1$ and width $N_2 - N_1 + 1$. For example, G5:G4 has height 0, while G6:G4 has height -1. Ranges of negative height or width are generally invalid, though our treatment will vary in exactly where we exclude them.

Formulas F are also standard in spreadsheets, and again Figure 2 shows our syntax. A formula can refer to a range using a *range reference* ρ, such as F4 or \$G\$2. (The superscript annotations in our syntax for ρ are explained in Section 3.4 and omitted until then.) Range *references* differ from ranges because they can be *relative* (e.g. B7) or *absolute* (e.g. \$B\$7), a choice that can be made independently for each axis of the reference (e.g. B\$7 or \$B7). During copy/paste, relative references are updated to reflect their new location, while absolute references remain unchanged.

The range on the left-hand side of a range assignment is called a *tile*. Tiles must have nonnegative height and width[8] and must not overlap, so that each cell is defined only once. A range assignment $t\ r = F$ means "put formula F into the top left-hand corner of r, and then use copy/paste to assign a formula to the other cells in r". So in our example

```
G4:G6 = F4 * $G$2
```

the cell G4 gets the original formula F4 $*$ \$G\$2, while G6 will get the formula F6 $*$ \$G\$2, as adjusted by copy/paste. The general rule is that when a formula is copy/pasted from cell C to D, its relative references are adjusted by the offset from C to D, while its absolute references are unchanged. The advantage of our range-assignment notation, compared to

[8] Tiles of height or width zero have no effect, but it is easier if we allow some of the algorithms later in the paper to generate Calculation View forms containing such tiles.

Column name[9]	$N ::= \mathrm{A} \mid \ldots \mid \mathrm{Z} \mid \mathrm{AA} \mid \mathrm{AB} \mid \ldots \mid \varnothing$
Row number	$m \in \mathbb{Z}^{\geq 0}$
Address (column, row)	$a ::= Nm$
Range	$r ::= a_1 : a_2$ (We write a short for $a : a$)
Absolute/relative marker	$\mu, \nu ::= \$ \mid$ ‘’
Cell reference	$\theta ::= \nu N \mu m$
Annotated range reference	$\rho ::= (\theta_1 : \theta_2)^{t_1,\ldots,t_n}$ (We write θ short for $\theta : \theta$)
Constant	c (either string or number)
Formula	$F ::= \rho \mid c \mid f(F_1, \ldots, F_n)$ (f either builtin function or an SDF)
Tile name	t
Range assignment	$\mathscr{A} ::= t\ r = F$
Sheet fragment	$\mathscr{S} ::= \mathscr{A}_1, \ldots, \mathscr{A}_n$
SDF definition	$\mathscr{F} ::= \text{function } f(t_1\ r_1, \ldots, t_k\ r_k) \text{ returns } r^{t'_1,\ldots,t'_j}\ \{\ \mathscr{S}\ \}$
Length variable	$\alpha, \beta, \ldots$
Elastic column name	$\overline{N} ::= N \mid \{N + \alpha\}$
Elastic row number	$\overline{m} ::= m \mid \{m + \alpha\}$
Elastic syntax elements	$\overline{a}, \overline{r}, \overline{\theta}, \overline{\rho}, \overline{F}, \overline{\mathscr{A}}, \overline{\mathscr{S}}, \overline{\mathscr{F}}$ (with N, m replaced by $\overline{N}, \overline{m}$)

Fig. 2: Spreadsheet syntax, in the style of Calculation View (Sarkar et al., 2018).

copy/paste in the grid, is that it makes explicit that the entire range shares a single master formula.

In the rest of the paper, when we are focusing on an individual range reference ρ within a formula for a tile t_c, we say that it can be *resolved* to a concrete range r, at a given cell c of t_c, by adjusting relative references for a copy/paste from the top-left corner of t_c to c and then discarding absolute/relative markers, which are no longer needed. We sometimes use the notation $r = \mathrm{Res}(\rho, c)$. Note that the top-left corner of t_c, from which the copy/paste starts, is implicit here. For example, resolving the relative range reference F4 (as it appears in the range assignment in the previous paragraph) at the cell G6 results in the concrete range F6, while resolving the absolute range reference \$G\$2 at G6 results in the concrete range G2. Resolution can also be applied to any piece of syntax that is part of a range reference, to produce the corresponding syntax that is part of a concrete range.

3.2 *Sheet-defined functions*

Next, we extend our Calculation View notation to cover sheet-defined functions (Figure 2). For example, we can abstract the re-usable computational content of the sheet fragment in Section 3.1 as an SDF, like this:

[9] We use $\varnothing$ to denote column number zero. Row and column zero do not exist in the sheet and we will ensure we do not actually access them, but they are needed in the syntax to express elastic ranges that start in the first row or column and can shrink to zero size, e.g., A1:A$\{0+\alpha\}$ and A1:$\{\varnothing+\alpha\}$1. Row and column zero are allowed only in the second endpoint of a range or range reference, not in the first endpoint; e.g., A1:A0 is a valid range with one column and zero rows, while A0 and A0:A2 are invalid as ranges.

```
function SHOP( F4:F6, G2 ) returns H7 {
 G4:G6 = F4 ∗ $G$2
 H4:H6 = F4 + G4
 H7 = SUM(H4:H6) }
```

The spreadsheet in Figure 1 contains the body of the SDF as well as two calls to it, the second of which requires elasticity in order to work correctly.

In general, an SDF definition $\mathscr{F}$ consists of a function name, a list of input ranges, an output range, and a set of range assignments that make up the body. The *body tiles* of an SDF are the left-hand sides of its range assignments; the *input tiles* are the SDF's input ranges. Each body tile has a single formula, namely the right-hand side of the range assignment. The order in which the body tiles are written is immaterial.

Intuitively, the semantics of a call F(e1,e2) to a sheet-defined function F is to evaluate the arguments to values v1 and v2, generate a fresh temporary spreadsheet containing the range assignments in F's body, initialise the input ranges with the argument values v1 and v2, calculate the value of each cell on the sheet (respecting dependencies), and return the value of the output range while discarding the temporary sheet. (We formalize this process in Section 5.2.) Unlike conventional languages, where the parameters of the function are (arbitrary) names given to the input values, in our language the input ranges specify the cell(s) in which the arguments to the function are placed; the output range specifies the cell(s) whose computed value is the result of the function. In our example, the first argument to SHOP is placed in F4:F6, while the second is placed in G2.

3.3 Assumptions on sheet-defined functions

For the sake of simplicity, we only attempt to generalise sheet-defined functions that are *tame* in the sense that they satisfy the following properties. We discuss whether these restrictions can be relaxed in Section 6.8, after we have covered enough about our system that the design possibilities are clear.

Static The range references that appear in a formula identify all the cells that are needed to evaluate the formula. In Excel the call INDIRECT(`"A2"` & `"3"`) first computes the string `"A23"`, and then treats it as a cell reference A23, so any use of INDIRECT makes a formula non-static. The function OFFSET is similar; but functions like INDEX and VLOOKUP are fine.

Closed For each range reference in the SDF and each cell in its calling tile, resolving the reference at that cell results in a range of non-negative height and width, and every cell in this range is inside one of the tiles of the SDF. Likewise, the output range has non-negative height and width, and every cell in it is inside one of the tiles of the SDF. For example, consider the tile

```
F4:F6 = SUM(H4:J4)
```

When copy/pasted into F5 the formula SUM(H4:J4) becomes SUM(H5:J5) and similarly for F6. Each cell in each of these ranges must be defined by some tile of the SDF.

No array tiles In many spreadsheet tools, it is possible to set an entire range equal to the array returned by a formula. We expect that spreadsheet tools that support SDFs may support such "array tiles" as part of SDFs, and the Calculation View notation of Sarkar et al. (2018) could naturally be extended to describe them, for example:

```
function SYMMETRIZE( A1:E5 ) returns A13:E17 {
  A7:E11 {=} TRANSPOSE(A1:E5)
  A13:E17 = (A1 + A7)/2 }
```

Generalising SDFs containing array tiles would be very useful. Unfortunately, array tiles present serious difficulties to our generalisation approach because their sizes may not be known statically in terms of the input sizes of the SDF. Hence, we do not include array tiles in the Calculation View syntax of this paper, so they cannot occur in tame SDFs.

Non-degenerate Each tile has positive height and width. Each range reference resolves to a range of positive height and width at some cell of the calling tile. Non-degeneracy makes our proofs simpler by avoiding corner cases, and we know of no useful functions that are thereby excluded.

Generalising a tame SDF and then specialising it to input sizes different from the originals will produce a *labelled SDF* (see the end of Section 4.3 for the precise definition) that may be degenerate (e.g., if some of the new input sizes are zero) but still satisfies the other properties.

3.4 Tiles and dependencies

To assign semantics (Section 4.2) to the elastic SDF that results from generalising a given tame SDF, we will find it helpful to first name each tile in the original SDF and to make explicit the tiles to which each reference points, like this:

```
function SHOP( $t_1$ F4:F6, $t_2$ G2 ) returns H7$^{t_5}$ {
  $t_3$ G4:G6 = F4$^{t_1}$ $*$ $G$2$^{t_2}$
  $t_4$ H4:H6 = F4$^{t_1}$ + G4$^{t_3}$
  $t_5$ H7 = SUM(H4:H6$^{t_4}$) }
```

We have given a distinct name, $t_1, t_2, \ldots$ to each of the tiles, which in this example consist of the three lines in the body of the function, together with the two input ranges. The *calling tile* of a range reference is the tile in whose formula that reference appears; for example, the calling tile of the range reference \$G\$2^{t_2} is t_3.

Furthermore, for each reference in the tile formulas, and in the returns, we have made explicit the tile(s) to which that reference points, using a superscript – these are the *target tiles* of the reference. A reference *points to* a target tile t if evaluating the reference (or any of the adjustments of the reference obtained by copy/paste in the same tile, if applicable) would read a cell from tile t. (More formally, a reference points to t if resolving it at some cell in the calling tile produces a range that overlaps t.) For example, the reference G4^{t_3} in the formula for tile t_4 is labelled with target tile t_3, because evaluating G4 would require the value of cell G4 which is in tile t_3. We can determine these annotations statically because the SDF is assumed to be both static and closed.

The output range in the returns clause plays a similar role to range references in tile formulas in that both are used to read data from the sheet, and accordingly, both are annotated with target tiles. Thus, to enable our algorithms to process the output "range reference" along with other range references, we pretend that each SDF contains a dummy 1-by-1 tile that serves as the calling tile of the output range reference. The output range reference is similar to a range reference in a formula for a 1-by-1 tile in that neither range reference is subject to adjustment of relative references; by introducing the dummy 1-by-1 tile, our algorithms will treat both kinds of range references the same way.

In the SHOP example, each reference has a *single* target tile, but in general, the label may be a *finite set of target tiles*. For example, suppose the final line of SHOP was instead

t_5 H7 = SUM(G4:H6^{t_3,t_4})

In this (contrived) example, the reference G4:H6 covers both tiles t_3 and t_4, and must be so labelled. We show a more realistic example in Section 8.

3.5 From spreadsheet grid to textual form

A textual form makes the technical development of a generalisation algorithm much easier, but that does *not* imply that we expect user to write textual function definitions in Calculation View form. Rather, we envisage three steps to get from a spreadsheet grid to a Calculation View definition of a SDF:

- *Identifying tiles*. Given a spreadsheet grid, we can readily identify its tiles, by identifying the largest rectangular ranges that share a common formula. One possible algorithm is given by Sarkar et al. (2018, Section IV.C). In most cases this decomposition into ranges is unique, but not in every case (e.g. think of an L-shaped range, which can decompose into rectangles in two different ways). Moreover, there are two ways to decide if two cells share a common formula: we might ask if their formulas came from copy/paste of a single original formula; or we might ask if independently-entered formulas are syntactically identical (modulo copy/paste). For the purposes of this paper, however, we assume that the decomposition is given.
- *Labelling*. Now we can add tile labels to every reference in the right hand side of a tile definition, simply by working out which tiles that reference can point to. This step is straightforward, given our restriction to static SDFs (Section 3.3).
- *SDF definitions*. From a pure grid we cannot infer the name, input cells, output cells, and boundary of an SDF. Instead we expect there will be direct UI support for delineating these aspects of an SDF's definition.

Although we regard the grid view as primary, we believe that users may actually benefit from an explicit textual calculation view, complementing the data-centric grid view offered by typical spreadsheets. We have some evidence that providing two views makes users more productive (Sarkar et al., 2018). But that thesis is orthogonal to this paper; here we simply assume that through some means (graphical, textual, whatever) the user has defined an SDF, and we propose a simple, concrete, textual notation as a formal tool in which we (not the user!) can write down that SDF and reason about it.

4 Elastic SDFs

The main focus of the paper is the task of *generalising* an SDF to an appropriate *elastic SDF*. We divide the process into three steps:

1. *Calculation View*. Construct a textual definition of the SDF in Calculation View notation, as discussed in Section 3.5.
2. *Generalisation.* Generalise the SDF to an elastic SDF $\overline{\mathscr{F}}$. This step is our main technical contribution, and is described in Sections 6 through 8.
3. *Code generation.* Transform the elastic SDF to executable form, which can be done in a variety of ways (Section 9).

In this section we introduce generalisation informally by way of an example (Section 4.1). We rely on this example as we introduce elastic SDFs: we describe the syntax and semantics of elastic SDFs (Section 4.2), and say what it means for an SDF to be well-defined (Section 4.3). We return to generalization in the following sections.

4.1 Generalisation by example

We begin with an example to illustrate the process. Suppose we start with the SHOP SDF introduced in Section 3.4. Step 1 is to annotate it with tiles, as described in Section 3.4, thus:

```
function SHOP( $t_1$ F4:F6, $t_2$ G2 ) returns H7$^{t_5}$ {
 $t_3$ G4:G6 = F4$^{t_1}$ $*$ $G$2$^{t_2}$
 $t_4$ H4:H6 = F4$^{t_1}$ + G4$^{t_3}$
 $t_5$ H7 = SUM(H4:H6$^{t_4}$) }
```

Next, in Step 2 we *generalise* this definition, to become an *elastic SDF*, thus:

```
function SHOP( $t_1$ F4:F{3+$\alpha$}, $t_2$ G2 ) returns H7$^{t_5}$ {
 $t_3$ G4:G{3+$\alpha$} = F4$^{t_1}$ $*$ $G$2$^{t_2}$
 $t_4$ H4:H{3+$\alpha$} = F4$^{t_1}$ + G4$^{t_3}$
 $t_5$ H7 = SUM(H4:H{3+$\alpha$}$^{t_4}$) }
```

We have introduced a *length variable*, α, which stands for the length of the input vector. A length variable can take any non-negative integer value. The size of the input range is α rows by 1 column, and the intermediate ranges rooted at G4 and H4 share this same size. (If for some reason we needed the ranges to be nonempty, we would just set their bottom rows to {4+α}.) The idea is, of course, that if we instantiate α to 3, we recover exactly the SDF that we started with. Notice that the tile labels are unaffected by generalisation.

For Step 3, we discuss what this elastic SDF means (its semantics) in Section 4.2, and how it might be executed in Section 9.

While the SHOP example illustrates a common spreadsheet layout practice in which all ranges whose height is the same length variable cover the same row span in the original SDF (rows 4 through 6), our system does not assume that this property holds. It would still work, for example, if tile t_3 were moved to G5:G7 (and references updated accordingly).

4.2 Elastic SDFs: syntax and informal semantics

As we have seen, elasticity requires us to generalise spreadsheet notation by allowing cell coordinates to be computed based on the function's length variables. The full syntax is given in Figure 2. We generalize column names N and row numbers m to include the possibility of adding a single length variable (enclosed in curly braces).[10]

For example, in the SHOP example, in tile t_5 H7 = SUM(H4:H$\{3{+}\alpha\}$), the right-hand side mentions a range reference H4:H$\{3{+}\alpha\}$ which has an elastic bottom row.

Analogous to our rule for inelastic SDFs, we require that every tile range have nonnegative height and width for all values of length variables. So, for example, a tile range of H4:H$\{1{+}\alpha\}$ is invalid because when $\alpha = 0$, it becomes H4:H1, which has height -2.

An elastic SDF is the central concept of the paper, so it needs a direct execution semantics. At first this looks straightforward: for example, to evaluate a call SHOP(e1, e2), using the elastic SDF resulting from Step 2 in Section 4.1:

- evaluate e1 and e2 to values v1 and v2;
- instantiate the body of SHOP with α equal to the number of rows in v1;
- compute the value of the return range using ordinary spreadsheet semantics.

But there is a tiresome problem: if $\alpha > 3$, then the ranges of tiles t_4 and t_5 *overlap*; for example, both t_4 and t_5 define a value for H7.

Disambiguating these collisions is the reason that we introduced tile names t_i, at both definition and use sites. They allow us to speak unambiguously of "the H7 defined by tile t_4" and "the H7 defined by tile t_5". For example, during evaluation, when dereferencing H7^{t_5} (in the returns position), we choose the value computed in H7 by tile t_5, ignoring any value for H7 by tile t_4, using the label attached to the reference H7^{t_5} to disambiguate which defining tile is intended. This semantics is easy to formalise, and we do so in the next section, Section 5.

4.3 Well defined elastic SDFs

Not every syntactically-correct elastic SDF is well defined according to this semantics. Two main things can go wrong.

- First, to be fully defined and unambiguous, the semantics requires that *when evaluating a reference, there should be a unique tile among the target-tile set labelling the reference that defines the referenced cell*, not zero (undefined) and not more than one (ambiguous). For the undefined case, consider this bogus SDF:

  function BOGUS_SHOP(t_1 F4:F$\{3{+}\alpha\}$, t_2 G2) returns H7^{t_5} {
  t_3 G4:G$\{3{+}\alpha\}$ = F4^{t_1} $*$ \$G\$2^{t_2}
  t_4 H4:H6 = F4^{t_1} + G4^{t_3} /* NB: Bogus! */
  t_5 H7 = SUM(H4:H$\{3{+}\alpha\}^{t_4}$) }

[10] We could allow adding a linear combination of length variables, but such combinations will never occur in the principal regular generalisation of an SDF as defined later, so we disallow them to save a little bit of worry about whether all the intervening definitions (e.g., determinability in Definition 1 below) make sense with linear combinations.

The trouble is that tile t_4 does not resize with its inputs t_1 and t_3. Consequently, if $\alpha < 3$, the formula for t_4 tries to read F6^{t_1}, but t_1 does not define F6. Similarly, if $\alpha > 3$, the same happens when the formula in t_5 tries to dereference H7^{t_4}.
In these cases, the uniqueness property fails because no target tile contains the referenced cell. But it can also happen that *too many* target tiles contain the cell. In our SHOP example, if the returns clause had been mis-labelled H7^{t_4,t_5}, the reference to H7 would be ambiguous because both t_4 and t_5 define H7 when $\alpha > 3$.
In contrast, the reference in the tile shown below (from the end of Section 3.4) is unambiguous, despite being labelled with two tiles:

```
$t_5$ H7 = SUM(G4:H6$^{t_3,t_4}$)
```

When evaluating the reference G4:H6^{t_3,t_4}, for every cell in the range G4:H6, there should be a unique tile among t_3, t_4 that defines the cell. For example, for G6, that tile is t_3. Similarly H5 is in that range, so it too should be defined by exactly one of the tiles t_3, t_4 – in this case t_4.

- Second, to have a well-defined semantics, an elastic SDF should not mention length variables that are not fixed by its inputs. For example:

```
function NONDET( A1:A{$\alpha$} ) returns B1 {
  C1:C{$\beta$} = ...
  B1 = SUM( C1:C{$\beta$} ) }
```

  Here β is not determined by the size of any of the input parameters, so it is hard to see how to execute the SDF.

These considerations motivate our definition of what it means for an elastic SDF to be well-defined. A *length assignment* for an elastic SDF $\overline{\mathscr{F}}$ is a function ϕ that maps each length variable appearing in $\overline{\mathscr{F}}$ to a nonnegative integer. Now we define:

Definition 1 (Well-defined elastic SDF)**.** *An elastic SDF $\overline{\mathscr{F}}$ is* well-defined *if it has the following properties:*

1. *It is* unambiguous*, meaning that under every length assignment for $\overline{\mathscr{F}}$:*
   a. *No two tiles in the target-tile set of the same reference overlap*[11].
   b. *For every range reference $\overline{\rho} = (\overline{\theta}_1 : \overline{\theta}_2)^{t_1,\ldots,t_k}$ in $\overline{\mathscr{F}}$*[12] *and every cell a in the calling tile of $\overline{\rho}$,* $\mathrm{Res}(\overline{\rho}, a)$ *has non-negative height and width and is covered by the tiles $t_1, \ldots, t_k$. (It is not required that every target tile of $\overline{\rho}$ actually overlap* $\mathrm{Res}(\overline{\rho}, a)$ *for some a; indeed, in Section 8.2, we will see an example elastic SDF in which some tiles in the target-tile set of a reference are "unused" for certain length assignments.)*
2. *It is* determinable*: all its length variables are uniquely determined by the sizes of its arguments. More formally, for any given list of argument sizes, there is at most one length assignment for $\overline{\mathscr{F}}$ under which the input tile sizes match the given argument sizes. (There could be none if, for example, $\overline{\mathscr{F}}$ requires that two parameters both*

[11] This condition may seem stronger than it needs to be. To unambiguously evaluate the reference, it is enough if no two target tiles overlap at any of the cells actually accessed by the reference. However, the stronger condition will be helpful for several of the code generation methods in Section 9.
[12] This includes the returns reference, which by our convention has a dummy calling tile.

have height α but the arguments have different heights. Such a call to $\overline{\mathscr{F}}$ would generate a runtime error.)

Having seen how we intend to use elastic SDFs, we should pause to clarify some definitions. Sections 3.1–3.3 discussed conventional spreadsheets and SDFs that lack tile names and target-tile labels, which we call *unlabelled*. Since we make limited use of them in the paper, we do not formalise their syntax; it is essentially that of Figure 2 without the labels. A *labelled SDF* or *elastic SDF* follows the syntax for $\mathscr{F}$ or $\overline{\mathscr{F}}$ in Figure 2 and all the semantic rules previously stated for unlabelled SDFs, except that tiles may overlap. The definition of labelled and elastic SDFs does not place any condition on the target-tile labels; while we will often want the target-tile labels to be such that the SDF is unambiguous, we will want to study labelled and elastic SDFs that we have not proven to be unambiguous. On the other hand, our generalisation process always starts with a labelled SDF generated from a tame unlabelled SDF as per Section 3.4, which we call a *tame labelled SDF*. We will make frequent use of the fact that a tame labelled SDF, unlike a labelled SDF in general, has nonoverlapping tiles and target-tile labels that contain exactly the tiles read by each reference (an even stronger condition than unambiguity, which allows target-tile labels to include unused tiles). From here through Section 8, when we refer simply to an "SDF" or a "tame SDF", we mean a labelled one.

5 Formal semantics of SDFs and elastic SDFs

In this section, we describe a formal semantics for our core spreadsheet language that includes SDFs and elastic SDFs. Next, in Section 6 and Section 7, we describe how to generalize from an example SDF to an elastic SDF.

5.1 Semantics of inelastic formulas

We begin by defining the value of a formula F in the context of a sheet fragment $\mathscr{S}$.

A *value* V is either a string or number c, or a 2D array: we write $\{c_{1,1},\dots,c_{1,n};\dots;c_{m,1},\dots,c_{m,n}\}$ for an $[m\times n]$ 2D array, with $m\geq 0$ rows and $n\geq 0$ columns (Figure 3). In common with most spreadsheet systems, our semantics supports arrays that arise as intermediate values in formulas (such as arguments or results of function calls), but whole arrays may not be stored in cells.[13]

We write γ for the *context* of evaluating a formula. The context includes the current sheet fragment $\mathscr{S}$, and bindings for any function parameters (Figure 3). We assume a partial function lookup(t,a,γ) that returns the value or formula stored in tile t at cell address a in context γ.

We give the full definition of contexts and lookup in Section 5.2, but first we give our semantics of formulas. The value $[\![F]\!]\gamma$ of an inelastic formula F given context γ is as follows:

$$[\![c]\!]\gamma = c$$

[13] Still, our prototype implementation—the basis of our user study—does support arrays-in-cells.

Value	$V ::= c \mid \{c_{1,1},\ldots,c_{1,n};\ldots;c_{m,1},\ldots,c_{m,n}\}$
Binding	$\mathscr{B} ::= t\ r = V$
Context	$\gamma ::= (\mathscr{S}, \mathscr{B}_1 \ldots \mathscr{B}_n)$

Fig. 3: Additional data structures for formal semantics

$$\begin{aligned}
[\![f(F_1,\ldots,F_n)]\!]\gamma &= [\![f]\!]([\![F_1]\!]\gamma,\ldots,[\![F_n]\!]\gamma)\\
[\![(\nu N\mu m : \nu' N\mu' m)^{t_1,\ldots,t_n}]\!]\gamma &= [\![F']\!]\gamma \quad \text{if there is } i \in 1..n \text{ such that lookup}(t_i, Nm, \gamma) = F'\\
&\qquad\qquad \text{and } [\![F']\!]\gamma \text{ is not an array}\\
[\![(\theta_1 : \theta_2)^{t_1,\ldots,t_n}]\!]\gamma &= \{c_{1,1},\ldots,c_{1,n};\ldots;c_{m,1},\ldots,c_{m,n}\}\\
&\quad \text{where } \theta_1 : \theta_2 \text{ has size } [m \times n] \neq [1 \times 1]\\
&\quad \text{and each } c_{i,j} = [\![(\theta_{i,j} : \theta_{i,j})^{t_1,\ldots,t_n}]\!]\gamma\\
&\quad \text{where each } \theta_{i,j} \text{ targets position } (i,j) \text{ in } \theta_1 : \theta_2
\end{aligned}$$

These recursive equations amount to a denotational semantics of formulas. The semantics is undefined in circumstances where a spreadsheet would return an error value, or if there is a cycle between a formula and its own value in the grid. We do not formally treat errors or cycles in our semantics, but it would be a standard application of domain theory. The four equations can be understood as follows:

- The first equation defines the semantics of a constant formula to be the constant itself.
- The second equation defines the meaning of a call to a function f. If f is a function with arity n, we assume $[\![f]\!]$ is a function from n-tuples of values to values to represent the semantics of f. We assume suitable definitions of $[\![f]\!]$ for each builtin function. For example, for the division operator, the meaning $[\![/]\!]$ is a function that given (c_1, c_2) returns c_1/c_2 if both values are numbers and $c_2 \neq 0$; otherwise it returns a suitable error string.
- The third equation applies to a singleton range reference $\nu N\mu m : \nu' N\mu' m$ that targets the cell with address Nm. The condition that $[\![F']\!]\gamma$ is not an array is how we enforce that only scalars can be stored in individual cells. Hence, an attempt to access an array from a cell is undefined.
- The fourth equation applies to a non-singleton range and returns an array of constants, each of which is computed using a recursive call to compute a singleton range. Saying that $\theta_{i,j}$ *targets* position (i, j) in $\theta_1 : \theta_2$ means that the cell addressed by $\theta_{i,j}$ is at position (i, j) in the range $\theta_1 : \theta_2$, where the top-left corner is position $(1,1)$.

The following two subsections define $[\![f]\!]$ when f is an SDF or an elastic SDF.

5.2 Semantics of SDFs

To represent the actual parameters passed to an SDF, we introduce a notion of *binding*, $\mathscr{B}$, of the form $t\ r = V$. A binding $t\ r = V$ means that the formal parameter r, labelled as tile name t, is bound to the actual parameter V. We can now complete the definition of context:

a *context* γ is a pair $(\mathscr{S}, \mathscr{B}_1 \dots \mathscr{B}_n)$, where $\mathscr{S}$ is an inelastic sheet fragment, $n \geq 0$, and each $\mathscr{B}_i$ is a binding.

Consider a function name f defined by an inelastic (labelled) SDF $\mathscr{F}$:

$$\texttt{function}\ f(t_1\ r_1, \dots, t_k\ r_k)\ \texttt{returns}\ r^{t'_1,\dots,t'_j}\ \{\ \mathscr{S}\ \}$$

Its meaning $[\![f]\!] = [\![\mathscr{F}]\!]$ is a function from k-tuples of values to values given as follows:

$$[\![\mathscr{F}]\!] = \lambda(V_1, \dots, V_k).[\![r^{t'_1,\dots,t'_j}]\!]\gamma \quad \text{where } \gamma = (\mathscr{S}, \mathscr{B}_1 \dots \mathscr{B}_k) \text{ and each } \mathscr{B}_i = (t_i\ r_i = V_i)$$

(Unlike V above that was a syntactic nonterminal representing a value, here each V_i is a parameter of the lambda that receives a value and passes it along to $\mathscr{B}_i$.)

Finally, to complete the semantics we need to specify how lookup operates on contexts. If $\gamma = (\mathscr{S}, \mathscr{B}_1 \dots \mathscr{B}_k)$, let lookup$(t, a, \gamma)$ be the value or formula given by:

- If $t\ r = F$ is one of the range assignments in $\mathscr{S}$, and address a falls within the range r, and a_s is the top-corner of r, then lookup$(t, a, \gamma) =$ copy(a_s, a, F), where copy(a_s, a, F) is the formula obtained by copying the formula F from source address a_s to destination address a.
- If $t\ r = c$ is one of the bindings in $\mathscr{B}_1 \dots \mathscr{B}_k$, and range $r = a : a$, then lookup$(t, a, \gamma) = c$.
- If $t\ r = \{c_{1,1}, \dots, c_{1,n}; \dots; c_{m,1}, \dots, c_{m,n}\}$ is one of the bindings in $\mathscr{B}_1 \dots \mathscr{B}_k$, and the size of r is $[m \times n]$ (that is, the size of the actual argument), and address a targets position (i, j) in range r, then lookup$(t, a, \gamma) = c_{i,j}$.
- Otherwise, lookup(t, a, γ) is undefined. This happens when a does not fall in the range of t.

In the first bullet point, we rely on copy(a_s, a_d, F), which returns the formula obtained by copying formula F from source cell a_s and paste to a destination cell a_d. This function implements the standard spreadsheet concept (Sestoft, 2014) of formula copy/paste, as outlined in Section 3.1: relative coordinates are updated by the offset but absolute coordinates are unchanged. Here are two examples based on formulas from the body of the SHOP function from Section 3.4:

- copy$(H4, H5, F4^{t_1} + G4^{t_3}) = F5^{t_1} + G5^{t_3}$. The row offset is one, so the two occurrences of the relative row coordinate 4 update to 5. The column offset is zero, so the two relative column coordinates F and G remain the same.
- copy$(G4, G5, F4^{t_1} * \$G\$2^{t_1}) = F5^{t_1} * \$G\$2^{t_1}$. The relative coordinates update as in the previous example, but the absolute row and column coordinates in $\$G\$2$ are unchanged in the copy of the formula at the new position.

Hence, if context γ holds the sheet in the body of the SHOP function from Section 3.4, then we have:

$$\begin{aligned}
\text{lookup}(t_3, G5, \gamma) &= F5^{t_1} * \$G\$2^{t_2} \\
\text{lookup}(t_4, H5, \gamma) &= F5^{t_1} + G5^{t_3}
\end{aligned}$$

As explained in Section 4.2, we need the t parameter to uniquely dereference range references in the semantics of elastic SDFs. Assuming $\mathscr{F}$ is unambiguous (in the sense of Definition 1 but without using a length assignment), every time the third equation in Section 5.1 is invoked, there will be a unique i such that lookup(t_i, Nm, γ) is defined.

5.3 Semantics of elastic SDFs

Consider a function name f defined by an elastic SDF $\overline{\mathscr{F}}$:

function $f(t_1\ \bar{r}_1, \ldots, t_k\ \bar{r}_k)$ returns $\bar{r}^{t'_1,\ldots,t'_j}$ $\{\ \overline{\mathscr{S}}\ \}$

Recall from Section 4.3 that a length assignment for $\overline{\mathscr{F}}$ is a function ϕ that maps each length variable appearing in $\overline{\mathscr{F}}$ to a nonnegative integer. If P is part of the syntax of $\overline{\mathscr{F}}$, such as a range $\bar{r}$, a sheet $\overline{\mathscr{S}}$, or $\overline{\mathscr{F}}$ itself, we let $\phi(P)$ denote the piece of inelastic syntax generated from P by replacing each occurrence of a length variable α with $\phi(\alpha)$.

The meaning $[\![f]\!] = [\![\overline{\mathscr{F}}]\!]$ of the elastic SDF is the function:

$$[\![\overline{\mathscr{F}}]\!] = \lambda(V_1,\ldots,V_k).[\![\phi(\overline{\mathscr{F}})]\!](V_1,\ldots,V_k)$$
$$\text{where } \phi \text{ is the length assignment, if any,}$$
$$\text{such that size of } \phi(\bar{r}_i) \text{ equals size of } V_i \text{ for each } i$$

In the final constraint, we refer to the *sizes* of inelastic ranges $\phi(\bar{r}_i)$ and values V_i. Let the size $[m \times n]$ of an inelastic range consist of the number m of rows and the number n of columns. The size of an array with m rows and n columns is simply $[m \times n]$, and the size of a constant is $[1 \times 1]$. Assuming $\overline{\mathscr{F}}$ is well-defined, it is determinable, so there is at most one ϕ satisfying the size constraint. If such a ϕ exists, then $\phi(\overline{\mathscr{F}})$ is an inelastic (labelled) SDF, and we can apply the semantics from the previous section. If not, then $f(V_1,\ldots,V_k)$ is undefined.

5.4 A detailed example

Consider our elastic SDF $\overline{\mathscr{F}}$ from Section 1:

function AVG(t_1 A1:A$\{\alpha\}$) returns B3^{t_4} $\{\ \mathscr{S}\ \}$

where the sheet fragment $\mathscr{S}$ is the following:

t_2 B1 = SUM(A1:A$\{\alpha\}^{t_1}$);
t_3 B2 = COUNT(A1:A$\{\alpha\}^{t_1}$);
t_4 B3 = B1^{t_2}/B2^{t_3}

Then its meaning $[\![$AVG$]\!]$ is the following:

$$\lambda(V_1).[\![\phi(\overline{\mathscr{F}})]\!](V_1)$$
$$\text{for some } \phi = (\alpha \mapsto l)$$
$$\text{where size of } \phi(A1{:}A\{\alpha\}) \text{ equals size of } V_1$$

Consider the following example, including an example call to AVG.

G1=5; G2=6; G3=7; G4=AVG(G1:G3)

Since the range G1:G3 evaluates to the array value $\{5;6;7\}$, evaluation of the formula in G4 boils down to the following call:

$$[\![\text{AVG}]\!](\{5;6;7\})$$

In terms of the expression above for the meaning $[\![\text{AVG}]\!]$, we have $V_1 = \{5;6;7\}$ whose size is $[3 \times 1]$, and the equation $\phi(A1{:}A\{\alpha\}) = [3 \times 1]$ determines that $\phi = (\alpha \mapsto 3)$, and so we get:

$$[\![\text{AVG}]\!](\{5;6;7\}) = [\![(\alpha \mapsto 3)(\overline{\mathscr{F}})]\!](\{5;6;7\})$$

The concrete SDF $(\alpha \mapsto 3)(\overline{\mathscr{F}})$ is

function AVG(t_1 A1:A3) returns B3^{t_4} { $\mathscr{S}_3$ }

where the sheet fragment $\mathscr{S}_3$ is the following:

```
$t_2$ B1 = SUM( A1:A3$^{t_1}$ );
$t_3$ B2 = COUNT( A1:A3$^{t_1}$);
$t_4$ B3 = B1$^{t_2}$/B2$^{t_3}$
```

The meaning $[\![(\alpha \mapsto 3)(\overline{\mathscr{F}})]\!]$ is the following function from 1-tuples of values to values:

$$[\![(\alpha \mapsto 3)(\overline{\mathscr{F}})]\!] = \lambda V_1.[\![B3^{t_4}]\!]\gamma \quad \text{where } \gamma = (\mathscr{S}_3, \mathscr{B}_1) \text{ and } \mathscr{B}_1 = (t_1\ A1 : A3 = V_1)$$

Hence, we can conclude that evaluation of the formula in G4 produces the value 6 as follows:

$$\begin{aligned}
[\![\text{AVG}]\!](\{5;6;7\}) &= [\![B3^{t_4}]\!]\gamma \quad \text{where } \gamma = (\mathscr{S}_3, \mathscr{B}_1) \text{ and } \mathscr{B}_1 = (t_1\ A1 : A3 = \{5;6;7\}) \\
&= [\![B1^{t_2}/B2^{t_3}]\!] \quad \text{because lookup}(t_4, B3, \gamma) = B1^{t_2}/B2^{t_3} \\
&= [\![B1^{t_2}]\!]\gamma/[\![B2^{t_3}]\!]\gamma \\
&= [\![SUM(A1{:}A3^{t_1})]\!]\gamma/[\![COUNT(A1{:}A3^{t_1})]\!]\gamma \\
&= 18/3 \\
&= 6
\end{aligned}$$

6 Principal and regular generalisations

As soon as we begin to speak of "generalising" an SDF, it is natural to ask whether there may be many possible generalisations and, if so, how we decide which one to pick. This question arises classically in type systems, where one typically proceeds as follows. First, one says what it means for a term to have a type. Next, one defines a generalisation order between types. Finally, one shows that every (typeable) term has a principal, or most-general, type; and gives an algorithm to find it. We will proceed analogously here:

1. We have already specified what it means for an elastic SDF $\overline{\mathscr{F}}$ to be *well-defined* (that is, both *unambiguous* and *determinable*, see Definition 1).
2. We give a generalisation ordering between elastic SDFs, and say what it means for $\overline{\mathscr{F}}$ to be a generalisation of an SDF $\mathscr{F}$ (Section 6.1).
3. We give examples of SDFs that have no principal well-defined generalisation (Sections 6.3–6.5). These examples motivate a new concept of a *regular* generalisation (Section 6.6).
4. We prove that every SDF has a principal regular generalisation and give an algorithm to find it.
5. We show that the principal regular generalisation is unambiguous, but in obscure cases might not be determinable; Section 6.2 discusses what to do in this case.

$$\text{Length substitution} \quad \phi ::= \varepsilon \mid \phi, \alpha \mapsto l \mid \phi, \alpha \mapsto \beta + l \quad (l \in \mathbb{Z}^{\geq 0})$$

Fig. 4: Length substitutions

Notice that only step (4) discusses the generalisation algorithm; the others are entirely free of algorithmic considerations.

6.1 The generalisation ordering

We start with Step (2). Recall that we have a SDF $\mathscr{F}$ and we seek its principal generalisation, an *elastic SDF* $\overline{\mathscr{F}}$ (see Figure 2). An elastic SDF is just like an inelastic one, but it possesses some length parameters α. So an SDF is just a special case of an elastic SDF with no length parameters.

As usual with generalisation orderings, we need to define the relevant kind of substitution, which is a *length substitution*, shown in Figure 4. A length substitution maps each length variable α to either a constant length l or an expression $\beta + l$, where l is a non-negative integer and β is a length variable. (Contrast with a length assignment, which maps each length variable to a constant.) As we did for length assignments in Section 5, if P is a piece of elastic syntax such as as a range $\bar{r}$ or an SDF $\overline{\mathscr{F}}$, we let $\phi(P)$ denote the result of substituting length variables according to ϕ in P; with a length substitution, $\phi(P)$ is another piece of elastic syntax.

Definition 2 (More general than). *An elastic SDF $\overline{\mathscr{F}}_1$ is more general than (or, equivalently, a generalisation of) $\overline{\mathscr{F}}_2$ if there exists a length substitution ϕ such that $\overline{\mathscr{F}}_2 = \phi(\overline{\mathscr{F}}_1)$.*

For example, ND4 below (which is studied further in Section 6.2) is more general than ND1, as witnessed by the length substitution $\{\alpha \mapsto \alpha, \beta \mapsto \beta, \gamma \mapsto 3\}$.

```
function ND4($t_1$ A1:A{$\alpha$}, $t_2$ A5:A{4+$\beta$}) returns A13$^{t_4}$ {
  $t_3$ A9:A{8+$\gamma$} = 1
  $t_4$ A13 = SUM(A9:A{8+$\gamma$}$^{t_3}$) }
function ND1($t_1$ A1:A{$\alpha$}, $t_2$ A5:A{4+$\beta$}) returns A13$^{t_4}$ {
  $t_3$ A9:A11 = 1
  $t_4$ A13 = SUM(A9:A11$^{t_3}$) }
```

The following proposition is not used anywhere else in the paper but suggests that our notion of "well-defined generalisation" is reasonable:

Proposition 1. *If an elastic SDF $\overline{\mathscr{F}}$ is a well-defined generalisation of an (inelastic) SDF $\mathscr{F}$, then $\overline{\mathscr{F}}$ is semantically equivalent to $\mathscr{F}$ on arguments that are the same sizes as the input tiles of $\mathscr{F}$.*

Proof Let $r_1, \ldots, r_n$ be the input ranges of $\mathscr{F}$, and let $\bar{r}_1, \ldots, \bar{r}_n$ be those of $\overline{\mathscr{F}}$. Since $\overline{\mathscr{F}}$ is a generalisation of $\mathscr{F}$, there is a length substitution ϕ such that $\phi(\overline{\mathscr{F}}) = \mathscr{F}$, and in particular, $\phi(\bar{r}_i) = r_i$. Since $\mathscr{F}$ is inelastic, ϕ must map every length variable in $\overline{\mathscr{F}}$ to a nonnegative integer. Now suppose $\overline{\mathscr{F}}$ is called on arguments $V_1, \ldots, V_k$, where V_i has the same size as r_i. According to the semantics given in Section 5.3, we construct a length substitution ϕ' such that for each i, the size of V_i equals the size of $\phi'(\overline{r}_i)$; then we form the inelastic SDF $\phi'(\overline{\mathscr{F}})$ and call it on the arguments.

Since $\overline{\mathscr{F}}$ is well-defined, it is determinable, so for each length variable α in $\overline{\mathscr{F}}$, α must affect either the height or the width of some input range $\overline{r}_i$ in $\overline{\mathscr{F}}$. We know $\phi'(\overline{r}_i)$ is the same size as V_i, which is the same size as r_i (by earlier assumption), but $r_i = \phi(\overline{r}_i)$. Since α affects the size of $\overline{r}_i$, it must be that $\phi'(\alpha) = \phi(\alpha)$. Since this is true for every α in $\overline{\mathscr{F}}$, we have $\phi'(\overline{\mathscr{F}}) = \phi(\overline{\mathscr{F}}) = \mathscr{F}$, so the call to $\overline{\mathscr{F}}$ is semantically equivalent to a call to $\mathscr{F}$, as claimed. ■

It would be lovely if every SDF had a most general (principal) well-defined generalisation. But it doesn't: due to several problems that we describe in the following subsections, there may be multiple incomparable ways to generalise an SDF to make a perfectly well-defined elastic SDF. So, when asked to generalise an SDF, which of these incomparable generalisations should the generalisation algorithm choose? We explain our approach in Section 6.6.

As we explain the decisions we have made about how SDFs should be generalised, please bear in mind that this work represents a first effort to generalise SDFs in accordance with user intent in a range of realistic cases. There is a long tail of cases in which our system does not give the result a user might want; one can argue about whether each is realistic. While we mention some of them, we leave to future work a systematic study of how our system might be changed to accommodate each of these cases and whether the benefit outweighs the cost in complexity and impact on other cases. If a given elastic SDF system does not produce the generalisation a user wants for a given SDF, the user may be able to modify the SDF in some way to get the desired result, or they may need to turn to another programming paradigm (such as VBA in the setting of Excel) to write a size-polymorphic function; either way, they may need assistance from someone with more programming experience.

6.2 Problem 1: under-constrained sizes

It is possible that the original SDF has a body tile whose size is not constrained to match that of any input tile. In this case there may be multiple well-defined generalisations that set the size of that tile in different ways. Here's an example that is unrealistic but demonstrates the general problem well:

```
function ND0($t_1$ A1:A3, $t_2$ A5:A7) returns A13$^{t_4}$ {
 $t_3$ A9:A11 = 1
 $t_4$ A13 = SUM(A9:A11$^{t_3}$) } /* 3 */
```

This SDF has the following well-defined generalisations:

```
function ND1($t_1$ A1:A{$\alpha$}, $t_2$ A5:A{4+$\beta$}) returns A13$^{t_4}$ {
 $t_3$ A9:A11 = 1
 $t_4$ A13 = SUM(A9:A11$^{t_3}$) } /* Always 3 */
function ND2($t_1$ A1:A{$\alpha$}, $t_2$ A5:A{4+$\beta$}) returns A13$^{t_4}$ {
 $t_3$ A9:A{8+$\alpha$} = 1
 $t_4$ A13 = SUM(A9:A{8+$\alpha$}$^{t_3}$) } /* Equal to the length of the first input */
function ND3($t_1$ A1:A{$\alpha$}, $t_2$ A5:A{4+$\beta$}) returns A13$^{t_4}$ {
```

```
t3  A9:A{8+β} = 1
t4  A13 = SUM(A9:A{8+β}^t3) }  /* Equal to the length of the second input */
```

None of these is more general than any of the others, yet all specialise to the original function when $\alpha = \beta = 3$. Which do we want? Even if we were to decide that ND2 or ND3 should be preferred to ND1 because they have a variable for the size of t_3 where ND1 has a constant (dubious when there is no evidence of user intent that t_3 resize with t_1 or t_2), that would still leave the ambiguity between ND2 and ND3.

Our solution is to drop the requirement that the generalisation be determinable, so that we can get this generalisation, which *is* principal (but not executable, since γ is not determined):

```
function ND4(t1 A1:A{α}, t2 A5:A{4+β}) returns A13^t4 {
  t3 A9:A{8+γ} = 1
  t4 A13 = SUM(A9:A{8+γ}^t3)  }  /* Value depends on γ! */
```

Now, in the rare cases where the principal generalisation is not determinable, we simply set the non-determined length variables to their initial values (that is, the value used in the original function written by the user) and issue a warning. That procedure will result in ND1 above.

While this workaround is arguably ugly, it is at least easy to understand: if our system finds no indication that a tile's size should depend on the input sizes, then the tile does not resize. Normally, we expect users to write SDFs so that the size of every tile is determined by the input sizes. In some cases, one could use heuristics to guess the user's intent and determine the size of a tile rather than let it default to fixed size. For example, a user might try to count the number of elements of the input like this:

```
function MYCOUNT0(t1 A1:A3) returns C1^t3 {
 t2  B1:B3 = 1
 t3  C1 = SUM(B1:B3^t2) }
```

Here t_2 is positioned alongside t_1, but there is no actual reference to t_1 that would require the tiles to have the same size, so the size of t_2 is undetermined in the principal generalisation:

```
function MYCOUNT0(t1 A1:A{α}) returns C1^t3 {
 t2  B1:B{β} = 1
 t3  C1 = SUM(B1:B{β}^t2) }
```

Presumably the user wants this generalisation:

```
function MYCOUNT0(t1 A1:A{α}) returns C1^t3 {
 t2  B1:B{α} = 1
 t3  C1 = SUM(B1:B{α}^t2) }
```

To get this result, we could change the system to require that adjacent tiles that have the same length in the original SDF have the same length variable in the elastic SDF. We do not consider this design possibility further in this paper. A workaround that works in this and many other examples is to force the tiles to have the same size by adding a dummy relative reference, e.g., by writing t_2 B1:B3 = 1 + 0∗A1^{t_1} in the original SDF.

6.3 Problem 2: arbitrary tile locations

The next problem is that tiles may be positioned in different ways as a function of the length variables, as long as the initial values of the variables give the initial positions. For instance, in the SHOP example, we could gratuitously make the column of the output cell depend on α:

```
function SHOP( $t_1$ F4:F{3+$\alpha$}, $t_2$ G2 ) returns {E+$\alpha$}7$^{t_5}$ {
  $t_3$ G4:G{3+$\alpha$} = F4$^{t_1}$ * $G$2$^{t_2}$
  $t_4$ H4:H{3+$\alpha$} = F4$^{t_1}$ + G4$^{t_3}$
  $t_5$ {E+$\alpha$}7 = SUM(H4:H{3+$\alpha$}$^{t_4}$) }
```

Neither the above nor the generalisation in Section 4.1 can be converted into the other by a substitution for α.

Arbitrary re-location of tiles in the elastic SDF is of no interest; it is a bit like α-renaming the binders of a lambda-term. The simplest way to avoid this possibility is to require that the upper-left corner of each tile of the elastic SDF be constant; that is, mention no length variables.[14] Tile t_5 above violates this because its top-left corner is at {E+α}7.

One can imagine useful elastic SDFs that violate this restriction, in which a variable upper-left corner is not arbitrary but is used to position one tile below or right of an elastic tile. For example:

```
function BLUR( $t_2$ A2:A{1+$\alpha$}) returns B1:B{1+$\alpha$}$^{t_4}$ {
  $t_1$ A1 = 0    /* black pixel at top */
  $t_3$ A{2+$\alpha$} = 0    /* black pixel at bottom */
  $t_4$ B1:B{1+$\alpha$} = (A1$^{t_1,t_2}$ + A2$^{t_2,t_3}$) / 2 }
```

Here the author's intent is, perhaps, to position cell A{2+α} immediately after the last cell of the input range A2:A{1+α}. Unfortunately, accommodating this pattern of use while retaining principality is much more complicated than simply lifting the restriction on upper-left corners. We leave this problem to future work.

Note that these remarks apply to *tile definitions*. For *references* we allow *both* corners of the range to be elastic. Consider, for example:

```
function SUM( $t_1$ A1:A{1+$\alpha$} ) returns B{1+$\alpha$}$^{t_3}$ {
  $t_2$ B1              = A1$^{t_1}$
  $t_3$ B2:B{1+$\alpha$}  = A2$^{t_1}$ + B1$^{t_2,t_3}$ }
```

Here the returned cell reference B{1+α}, a shorthand for the range reference B{1+α}:B{1+α}, points to the bottom of tile t_3 and we want it to move down as α grows. This need to elasticise the top row of a reference really only applies when the reference is to a single row (an analogous statement holds for columns), and then only when using an enhanced version of the system to be described in Section 8 that supports references to the bottom row or rightmost column of a tile, but it is harmless to always allow elasticity in both corners of every reference.

[14] After generalisation is complete, one possible execution scheme might re-introduce length variables in the top-left corner to avoid overlaps – see Section 9.

6.4 Problem 3: generalising size-1 axes

Suppose we were to generalise a tile of height 1 to variable height α. We may then have a choice to interpret a reference to it as aggregating it or mapping over it, both of which are among the most common kinds of computation that we want to support. For example, this SDF:

```
function G(t₁ A1) returns B1^t₂ {
  t₂ B1 = COUNT(A1^t₁) }  /* Always returns 1 */
```

has the following possible incomparable generalisations:

```
function G(t₁ A1:A{α}) returns B1^t₂ {
  t₂ B1 = COUNT(A1:A{α}^t₁) }  /* Returns length of the input */
function G(t₁ A1:A{α}) returns B1:B{α}^t₂ {
  t₂ B1:B{α} = COUNT(A1^t₁) }  /* Returns vector of ones */
```

Our solution is to ban generalisation of a height of 1 to variable height, and similarly for width 1. This seems entirely reasonable: if the user wants an SDF to be generalised to an array of arbitrary size, she should write an example SDF that has an array of at least size 2, not size 1. With this rule, G has no further generalisation at all.

6.5 Problem 4: patterns of computation

The last problem is the trickiest: there may be multiple well-defined ways to elasticise the same reference. For example, the following SDF:

```
function F(t₁ A1:A2) returns B1:B2^t₂ { t₂ B1:B2 = A1^t₁ }
```

has the following incomparable generalisations[15]:

```
function F1(t₁ A1:A{α}) returns B1:B{α}^t₂ {
   t₂ B1:B{α} = A1^t₁ }  /* Returns the entire input */

function F2(t₁ A1:A{2+α}) returns B1:B2^t₂ {
   t₂ B1:B2 = A1^t₁ }  /* Returns first two elements of input */

function F3(t₁ A1:A{2+α}) returns B1:B2^t₂ {
   t₂ B1:B2 = A{1+α}^t₁ }  /* Returns last two elements of input */
```

The latter two generalisations are clearly a bit ad-hoc, because they pick two elements out of a variable-height input array, so the first is probably the generalisation that the user intended — but how should we formalise that intuition? We do so by requiring each reference in the elastic SDF to be *well-behaved*. It will turn out that the reference in F1 is well-behaved but those in F2 and F3 are not, resolving the ambiguity and making F1 the principal generalisation.[16]

[15] Even assuming that we adopt the choice in Section 6.4 and refrain from generalising the size-1 columns of the ranges.

[16] One could argue that this subsection 6.5 and the previous one 6.4 tackle aspects of the same problem (a variety of possible generalisations) with the same general solution (limit the allowed patterns of computation). However, as far as we know, the simplest way to achieve principality is to impose *both* the restriction on generalising size-1 axes (Section 6.4) and the well-behavedness condition for references (Section 6.5), with the generalisation algorithm having logic related to each condition. We do not believe there is a way to simplify the system by merging the two conditions.

The notion of well-behavedness captures the patterns of computation that our elastic SDFs can support. The full definition of well-behavedness that we want to use is rather complex since it seeks to support many patterns of computation. To make the paper easier to understand, we first describe the entire system using a simpler definition and then, in Section 8, we describe the necessary changes to use the full definition.

The simpler definition supports three kinds of references, all of which are exhibited in our SHOP example from Section 4.1, all of whose references are well-behaved:

```
function SHOP( $t_1$ F4:F{3+$\alpha$}, $t_2$ G2 ) returns H7$^{t_5}$ {
  $t_3$ G4:G{3+$\alpha$} = F4$^{t_1}$ * $G$2$^{t_2}$
  $t_4$ H4:H{3+$\alpha$} = F4$^{t_1}$ + G4$^{t_3}$
  $t_5$ H7 = SUM(H4:H{3+$\alpha$}$^{t_4}$) }
```

The three kinds are:

- *Inelastic.* The reference \$G\$2^{t_2} in the formula for t_3 can be copied down the range G4:G{3+α}, and always refers to the input cell G2. Similarly H7^{t_5} in the returns clause refers to a size-one range, and is not copied anywhere. We describe both these references as Inelastic.
- *Lockstep.* The reference F4^{t_1} in the formula for t_3 refers to the top cell of t_1; as it is copied down the range G4:G{3+α} it will refer to successive cells of the input range F4:F{3+α}; and those two ranges are the same size. The same applies to the references F4^{t_1} and G4^{t_3} in the formula for t_4. We describe these references as Lockstep.
- *Whole.* The reference H4:H{3+α}^{t_4} in the formula for t_5 refers to the whole of tile t_4, and is not copied anywhere (since t_5 is of unit height). We describe this reference as Whole.

Returning to the F example, we find that the reference A1^{t_1} in F1 is well-behaved because it follows the Lockstep kind. In contrast, the reference A1^{t_1} in F2 falls in none of these categories, and hence is not well-behaved. It is closest to Lockstep, because it is copied down the range B1:B2, but that is a fixed range of size 2, whereas the target tile t_1 has incompatible height $2+\alpha$. Similarly, the reference in F3 is not well-behaved.

We give a formal definition of well-behavedness in Section 6.9. But our goal is:

- Well-behavedness should "make sense"; that is, it should embody patterns of computation that users want. The more restrictive our definition of well-behavedness, the more limited is the expressiveness of our SDFs.
- Well-behavedness should cut down the space of generalisations so that a principal generalisation exists. This is a matter of proof.

Our initial definition of well-behavedness applies only to generalisations of *basic* SDFs, defined as follows:

Definition 3 (Basic SDF)**.** *A tame SDF is* basic *if each reference has only one target tile.*

We note that target-tile labels are unaffected by generalisation, so in a generalisation of a basic SDF, each reference still has only one target tile.

All the SDFs we have seen so far are basic. The extended definition of well-behavedness in Section 8 handles arbitrary tame SDFs (among other enhancements), and that section includes a realistic non-basic SDF as a motivating example.

6.6 Regularity

Due to the problems we have seen in the previous subsections, an SDF may not have a principal well-defined generalisation. We recover principality like this:

- We solve the first problem (Section 6.2) as described in that section, by finding a principal generalisation that may not be determinable, and making it determinable afterwards.
- We solve the remaining problems (Sections 6.3, 6.4, and 6.5) by restricting the set of generalisations among which we choose, to the *regular* ones (defined shortly), which obey the restrictions discussed in those sections.

The following definitions make the above outline precise.

Definition 4 (Regular generalisation)**.** *An elastic SDF* $\overline{\mathscr{F}}$ *is a* regular generalisation *of a basic SDF* $\mathscr{F}$ *if it satisfies the following conditions:*

1. $\overline{\mathscr{F}}$ *is a generalisation of* $\mathscr{F}$*.*
2. *The upper-left corner of each tile of* $\overline{\mathscr{F}}$ *is constant (Section 6.3).*
3. *Every tile of non-constant height in* $\overline{\mathscr{F}}$ *has height at least 2 in* $\mathscr{F}$*, and likewise for the width (Section 6.4).*
4. *Each range reference* $\overline{\rho}$ *in* $\overline{\mathscr{F}}$ *is well-behaved (as sketched in Section 6.5 and defined formally in Section 6.9).*

Definition 5 (Principal regular generalisation)**.** *Let* $\mathscr{F}$ *be a basic SDF. An elastic SDF* $\overline{\mathscr{F}}^*$ *is the* principal *regular generalisation of* $\mathscr{F}$ *(up to renaming of length variables) if it is a regular generalisation of* $\mathscr{F}$ *and is more general than every other regular generalisation of* $\mathscr{F}$*.*

Notice that we have not yet proved that a principal regular generalisation of $\mathscr{F}$ exists. In Section 7, we will give (and prove correctness of) an algorithm to find it, thereby proving that it exists.

6.7 Omitted features: function argument consistency and height/width coupling

A further requirement one could consider placing on regular generalisations that may help to honor user intent, though it does not affect the existence of a principal regular generalisation, is that the sizes of the arguments to a function call be consistent with what the function accepts. Such a requirement may allow the sizes of additional body tiles to be determined or may constrain the parameter sizes of the elastic SDF so that mismatched argument sizes can be caught up front. For an example of determining the size of an additional body tile, consider:

```
function WEIGHTEDSUM($t_1$ A2:A11) returns C1$^{t_4}$ {
  $t_2$ B1 = 0
  $t_3$ B2:B11 = B1$^{t_2,t_3}$ + 1
  $t_4$ C1 = SUMPRODUCT(A2:A11$^{t_1}$, B2:B11$^{t_3}$) }
```

In the current system[17], the principal generalisation would be:

```
function WEIGHTEDSUM($t_1$ A2:A$\{1+\alpha\}$) returns C1$^{t_4}$ {
  $t_2$ B1 = 0
  $t_3$ B2:B$\{1+\beta\}$ = B1$^{t_2,t_3}$ + 1
  $t_4$ C1 = SUMPRODUCT(A2:A$\{1+\alpha\}^{t_1}$, B2:B$\{1+\beta\}^{t_3}$) }
```

where β is not determined. According to Section 6.2, we would set β back to 10 and t_3 would not resize with the input. To solve the problem, since SUMPRODUCT requires two arrays of equal size, we could constrain its arguments to have equal size, and then we would get the desired generalisation:

```
function WEIGHTEDSUM($t_1$ A2:A$\{1+\alpha\}$) returns C1$^{t_4}$ {
  $t_2$ B1 = 0
  $t_3$ B2:B$\{1+\alpha\}$ = B1$^{t_2,t_3}$ + 1
  $t_4$ C1 = SUMPRODUCT(A2:A$\{1+\alpha\}^{t_1}$, B2:B$\{1+\alpha\}^{t_3}$) }
```

Generating constraints for a function call is a bit complicated in general because the arguments can be arbitrary expressions, which may include function calls whose result sizes cannot be statically determined in terms of the argument sizes. We do not formalise the rules to deal with this as it would distract from the main ideas of the paper. Furthermore, users may be surprised by the inconsistent behavior if we generate constraints when the argument sizes are statically known and silently skip generating constraints when the sizes are not known; we are unsure how best to address this issue. In the meantime, the workaround of forcing tiles to have the same size by adding a dummy relative reference (end of Section 6.2) works in some cases, including WEIGHTEDSUM.

Without constraints for function argument sizes, we find that the definition of a regular generalisation is completely independent in the vertical and horizontal directions, and no length variable will appear in both a row number and a column number in the principal regular generalisation. There is nothing that can be written in the original SDF to force generalisations to set the height of one tile equal to the width of another tile. This is because the only construct that couples the sizes of two tiles is a relative reference of the Lockstep kind, and when a relative reference is copy/pasted, the target cell always moves in the same direction as the caller cell, so the sizes of the caller and target tiles are coupled on the same axis. Consequently, one cannot reimplement a function like TRANSPOSE because there is no way to force the size of the output tile to generalise correctly.

The problem is only with determining tile sizes. If appropriate tile sizes are given, one can look up data from a row corresponding to the calling column or vice versa using the INDEX function. Here is an example in an inelastic SDF:

```
function MYTRANSPOSE($t_1$ A1:E7) returns A9:G13$^{t_2}$ {
  $t_2$ A9:G13 = INDEX($A$1:$E$7$^{t_1}$, COLUMNS($A9:A9$^{t_2}$), ROWS(A$9:A9$^{t_2}$))
}
```

[17] Actually, we need the system of Section 8 to handle the iterative computation in t_3. The difference is not important to the present discussion.

One could consider adding built-in functions that capture these use cases of INDEX and couple tile widths and heights appropriately on generalisation. We leave this minor extension to future work.

6.8 Could the restrictions on tame SDFs be relaxed?

Now that the reader has a sense of how generalisation works, we can consider whether any of the "tameness" restrictions on the original SDF from Section 3.3 could reasonably be relaxed.

Static We have no way to know what constraints should be imposed on an elastic SDF for a call to INDIRECT to function as the user intended, nor does it come with a target-tile set. Still, we can attempt to execute it by reading the requested cell, failing if more than one tile in the elastic SDF defines it or returning a blank if none does. If the user establishes any needed tile size relationships using dummy relative references (see the end of Section 6.2), it may be possible to use INDIRECT reliably. One could consider introducing a variant of INDIRECT that takes a target-tile set. Similar remarks apply to OFFSET($\rho, \ldots$), except in that case we have the option of using the target-tile set of ρ, which is guaranteed to have no overlap. We have not investigated whether users would want the target-tile set of ρ to be used, although we note that the behavior in which it is used can alternatively be achieved using INDEX in some cases.
There is a case to be made for supporting INDIRECT and OFFSET in elastic SDFs, based on their popularity with users. These functions are among the 15 most frequently used in the Enron and EUSES corpora, according to an analysis by Jansen (2015). However, there is also a case against including them: the behaviour of these functions is hard to reason about even in ordinary circumstances, and they would be extraordinarily hard to debug if they were acting from within an SDF call, especially as there are multiple possible schemes for implementing these functions and how they elasticise.

Closed If we allow non-closed references in the original SDF, we will have to allow them in elastic SDFs too. Our current definition of regularity serves the dual purpose of limiting patterns of computation to guarantee a principal regular generalisation and ensuring that closed references in the original SDF remain closed in the elastic SDF. One could conceivably develop a definition for non-closed SDFs that does the former without the latter, but we have no familiarity with the use cases (if any) for non-closed SDFs to suggest whether a *useful* definition of regularity could be developed. We leave this question to future work, if desired.

No array tiles Array tiles can potentially be handled by describing their sizes with length variables that are determined during SDF execution. Thus, the execution semantics for an elastic SDF can no longer determine the length variables from the argument sizes and simply specialise the elastic SDF to an inelastic one, but must work directly on the elastic SDF, interleaving formula evaluation with the determination of length variables from array tiles. Tricky problems arise when other parts of the SDF force the heights or widths of several array tiles to be the same length variable δ. When we need the value of δ, which of these array tiles t do we try to evaluate? If we find that t cannot be evaluated without already knowing δ, do we try to determine δ from another array tile? What if we find out later that different array tiles determine contradictory values of δ? We believe there are likely to be adequate solutions to these design problems, but we leave the details to future work. A further issue is whether we attempt to statically analyse array tile formulas to generate constraints on length variables; the same remarks apply as for consistency of function argument sizes in Section 6.7.

Non-degenerate We see no value in relaxing this restriction.

6.9 Complete definition of well-behaved reference

We now give the complete definition of a well-behaved reference that was sketched in Section 6.5 and will be used in the rest of the paper until Section 8. Aside from formalising the conditions for each kind of reference (including some variants that are harmless and useful to support even though they do not appear in the SHOP example), a major new issue we need to address is elasticity in the horizontal direction as well as the vertical direction. Indeed, we want to treat the two "axes" of the spreadsheet symmetrically. To this end, we number the columns starting from 1 like the rows, and we refer to either a row number or a column number as an *axis position*. Consistent with the previous syntax, an *elastic axis position* is an axis position, optionally with a single length variable added.

Although a range is traditionally given by a pair of corners, to allow ranges to adjust independently in the horizontal and vertical directions, it's more useful to think of a range as the intersection of a *row span* and a *column span*. Each span has two endpoints that are axis positions (elastic or not, as the context requires). For example, the range F4:G6 is the intersection of the row span 4:6 and the column span 6:7.

Recall that a range reference differs from a range in that each row and column has an absolute/relative marker. Thus, we think of a range reference as the intersection of a *row span reference* and a *column span reference*, where each endpoint of a span reference is an *axis position reference* that consists of an axis position with an absolute/relative marker μ. We review the syntax for the new concepts below. (For each concept, the non-elastic version simply requires that no length variables be used.)

Axis position	$m \in \mathbb{Z}^{\geq 0}$
Elastic axis position	$\overline{m} ::= m \mid \{m + \alpha\}$
Elastic axis position reference	$\overline{\chi} ::= \mu\overline{m}$
Elastic span	$\overline{s} ::= \overline{m}_1 : \overline{m}_2$
Elastic span reference	$\overline{\sigma} ::= \overline{\chi}_1 : \overline{\chi}_2$

By the *coordinate* of an elastic axis position reference $\overline{\chi} = \mu\overline{m}$, we mean the underlying elastic axis position $\overline{m}$ without regard to whether the reference is absolute or relative. When we say an elastic row reference $\overline{\chi}$ *points to* a given location in a tile, we mean that the coordinate of $\overline{\chi}$ equals the row number in the tile as an expression in terms of length variables and not just for the values of length variables that reproduce the original SDF. However, this condition applies only to $\overline{\chi}$ as written in the formula for the upper-left cell of the calling tile; it is understood that copy/paste may change the reference used to evaluate the remaining cells of the calling tile, unless the context rules this out.

Finally, we are ready to define a well-behaved reference:

Definition 6 (Well-behaved reference)**.** *Let $\overline{\mathscr{F}}$ be a generalisation of a basic SDF $\mathscr{F}$ (so every range reference in $\overline{\mathscr{F}}$ has a single target tile). A range reference $\overline{\rho}$ in $\overline{\mathscr{F}}$ is* well-behaved *if it satisfies the following conditions:*

1. *Let t_c be the calling tile and t_t be the target tile of $\overline{\rho}$. Let $\overline{\sigma} = \mu_1\overline{m}_1 : \mu_2\overline{m}_2$ be the row span reference of $\overline{\rho}$. Then $\overline{\sigma}$ satisfies the conditions of one of the following three kinds:*
   - Inelastic*: The height of t_t is constant, $\overline{m}_1$ and $\overline{m}_2$ are both constant, and either the height of t_c is constant or μ_1 and μ_2 are both absolute. Thus, $\overline{\sigma}$ and any copy/pasted variants will always point to the same row(s) of t_t as they did in $\mathscr{F}$.*
   - Lockstep*: The heights of t_c and t_t are the same non-constant expression, $\overline{m}_1 = \overline{m}_2$, $\mu_1 = \mu_2$ is relative, and $\overline{m}_1$ points to the top row of t_t. Thus, when $\overline{\sigma}$ is copy/pasted through t_c, each row of t_c points to the corresponding row of t_t.*
   - Whole*: The height of t_t is non-constant, $\overline{m}_1$ points to the top row of t_t, $\overline{m}_2$ points to the bottom row of t_t, and either the height of t_c is 1 or both μ_1 and μ_2 are absolute. Thus, $\overline{\sigma}$ always refers to the whole row span of t_t.*
2. *The analogous condition for the column span reference of $\overline{\rho}$.*

6.10 Unambiguity of a regular generalisation

At the beginning of Section 6, we claimed that the principal regular generalisation of an SDF $\mathscr{F}$ is unambiguous; in fact, this is true for any regular generalisation. We are finally ready to prove that fact.

Theorem 1. *Every regular generalisation $\overline{\mathscr{F}}$ of a basic SDF $\mathscr{F}$ is unambiguous.*

Proof We must show two conditions for each range reference $\overline{\rho}$ in $\overline{\mathscr{F}}$, which is well-behaved by our assumption that $\overline{\mathscr{F}}$ is regular. Since $\mathscr{F}$ is basic, $\overline{\rho}$ has only one target tile t_t, so there is no possibility of overlap among target tiles of $\overline{\rho}$. (Things will get more interesting when we modify this proof for non-basic SDFs in Section 8.) It remains to show that for every length assignment and at every cell in the calling tile t_c, $\overline{\rho}$ resolves to a range r of non-negative height and width that is covered by t_t. We show that r has non-negative height and its row span is contained in the row span of t_t; the argument for the column axis is analogous. Let $\overline{\sigma} = \overline{\chi}_1 : \overline{\chi}_2$ be the row span reference of $\overline{\rho}$.

- If $\overline{\sigma}$ is of the Inelastic kind, then t_t is fixed at its height in $\mathscr{F}$ and $\overline{\chi}_1$ and $\overline{\chi}_2$ contain no length variables. We also know that either $\overline{\chi}_1$ and $\overline{\chi}_2$ are both absolute (in which case $\overline{\sigma}$ always resolves to the same row span it did in $\mathscr{F}$) or t_c is fixed at its height in $\mathscr{F}$ (in which case, for each caller row, $\overline{\sigma}$ resolves to the same row span as it did for the same caller row in $\mathscr{F}$). Either way, since $\mathscr{F}$ is closed, it follows that for every caller row, $\overline{\sigma}$ resolves to a row span of non-negative height that is contained in the row span of t_t.
- If $\overline{\sigma}$ is of the Lockstep kind, then for every length assignment, t_t and t_c have the same height, and $\overline{\sigma}$ resolved at the ith row of t_c points to the ith row of t_t, which is indeed a row span of non-negative height that is contained in the row span of t_t.

Delta variable	$\hat{\alpha}, \hat{\beta}$
Shrinkage value	$l \in \mathbb{Z}^{\geq 0}$
Constraint	$Q \ ::= \ \hat{\alpha} = \hat{\beta} \ \mid \ \hat{\alpha} = 0 \ \mid \ \hat{\alpha} \geq -l$

Fig. 5: Delta variables and constraints

- If $\overline{\sigma}$ is of the Whole kind, then for every length assignment, $\overline{\sigma}$ resolves to the whole row span of t_t, which has non-negative height and is contained in itself.

This completes the proof. ∎

7 Elasticity inference

Next, we turn our attention to the task of finding the principal regular generalisation of an SDF $\mathscr{F}$. Our approach is quite conventional: introduce variables, generate constraints on those variables, and find their principal solution.

The variables we use are *delta variables*, denoted $\hat{\alpha}$, each of which represents the difference between an axis position (part of a tile range or range reference) in the elastic SDF and the original SDF. Setting all delta variables to 0 recovers the original SDF. Unlike length variables, delta variables can take negative values to allow tiles to shrink compared to the original SDF, though the amount of shrinkage will be limited by the need to keep tile sizes nonnegative and references within tile bounds.

The forms of constraints that we use are shown in Figure 5: namely, that two parts of the SDF move together, that one part not move at all, or that one part not shrink more than a certain amount. Note that no constraint of these forms can prevent us from recovering $\mathscr{F}$ by setting all delta variables to 0.

7.1 Overview of the inference algorithm

The elasticity inference algorithm is as follows:

1. *Elasticise.* Generate a "master" elastic version $\overline{\mathscr{F}}_0$ of $\mathscr{F}$ that possesses every degree of freedom that might conceivably be found in a regular generalisation of $\mathscr{F}$.[18] Specifically:

   – *Tile definitions*. Add a fresh delta variable $\hat{\alpha}$ to the bottom row and right column of each tile definition. So a tile definition (e.g. the LHS of a range assignment) t G4:H6 becomes t G4:{H+$\hat{\alpha}_1$}{6+$\hat{\alpha}_2$}. The same applies to the tile definitions in the arguments of the SDF. Note that we elasticise only the bottom-right corner of each tile (on the LHS), not the upper-left corner, to respect item 2 of Definition 4.[19]

[18] Strictly speaking, $\overline{\mathscr{F}}_0$ is not an elastic SDF because it contains delta variables that may take negative values. Its syntax is that of $\overline{\mathscr{F}}$ in Figure 2 with delta variables in place of length variables.

[19] The dummy calling tile of the returns reference gets variables added in the same way, although they will just be constrained to 0 in step 2a.

- *References*. Add a fresh delta variable to each component of each corner of each reference in a formula. So a reference C3:D7 becomes $\{$C+$\hat\alpha_1\}\{3+\hat\alpha_2\}$:$\{$D+$\hat\alpha_3\}\{7+\hat\alpha_4\}$. The same applies to the reference in the returns clause of the SDF.

Note that we elasticise the bottom-right corner of a tile definition, but both corners of a reference, for reasons discussed in Section 6.3.

2. *Generate constraints.* Generate a set of constraints on the delta variables, in the syntax given in Figure 5, as follows:

   a. For each tile, if the height was 1 in $\mathscr{F}$, then constrain the bottom-row delta variable equal to 0[20]; otherwise constrain the bottom-row delta variable so as to ensure that the height of the tile is non-negative. Do likewise with the width.
   b. Generate constraints for each range reference $\overline{\rho}$ in $\overline{\mathscr{F}}_0$, as described in Section 7.3.

3. *Solve constraints.* Find the principal solution of the constraints, a *delta substitution* Θ^* that maps every delta variable in $\overline{\mathscr{F}}_0$ to either zero or $\alpha - l$ where α is a length variable and $l \in \mathbb{Z}^{\geq 0}$ (see Figure 6). The constraint solving process is described in Section 7.2.
4. *Extract the principal solution.* Apply Θ^* to $\overline{\mathscr{F}}_0$ to produce the principal regular generalisation $\overline{\mathscr{F}}^*$ of $\mathscr{F}$.

We illustrate the algorithm using the SHOP SDF introduced in Section 3.4. We start from this SDF:

function SHOP(t_1 F4:F6, t_2 G2:G2) returns H7:H7^{t_5} {
  t_3 G4:G6 = F4:F4^{t_1} $*$ \$G\$2:\$G\$2^{t_2}
  t_4 H4:H6 = F4:F4^{t_1} + G4:G4^{t_3}
  t_5 H7:H7 = SUM(H4:H6^{t_4}) }

Then we take the following steps:

1. Here is the master elastic version $\overline{\mathscr{F}}_0$ (to reduce clutter, we have omitted the delta variables for the columns since nothing interesting happens there):

   function SHOP(t_1 F4:F$\{6+\hat\alpha_1\}$, t_2 G2:G$\{2+\hat\alpha_2\}$)
     returns H$\{7+\hat\alpha_3\}$:H$\{7+\hat\alpha_4\}^{t_5}$ {
    t_3 G4:G$\{6+\hat\alpha_5\}$ = F$\{4+\hat\alpha_6\}$:F$\{4+\hat\alpha_7\}^{t_1}$ $*$ \$G\$$\{2+\hat\alpha_8\}$:\$G\$$\{2+\hat\alpha_9\}^{t_2}$
    t_4 H4:H$\{6+\hat\alpha_{10}\}$ = F$\{4+\hat\alpha_{11}\}$:F$\{4+\hat\alpha_{12}\}^{t_1}$ + G$\{4+\hat\alpha_{13}\}$:G$\{4+\hat\alpha_{14}\}^{t_3}$
    t_5 H7:H$\{7+\hat\alpha_{15}\}$ = SUM(H$\{4+\hat\alpha_{16}\}$:H$\{6+\hat\alpha_{17}\}^{t_4}$) }

2. Generate constraints. Here we show $\overline{\mathscr{F}}_0$ again, with each constraint attached to the part of $\overline{\mathscr{F}}_0$ it was generated from. Constraints generated for range references are marked with the step of the procedure in Section 7.3 that generated them. Unmarked constraints are from step 2a of the main algorithm at the beginning of this section.

   function SHOP(t_1 F4:F$\{6+\hat\alpha_1\}^{[\hat\alpha_1 \geq -3]}$, t_2 G2:G$\{2+\hat\alpha_2\}^{[\hat\alpha_2 = 0]}$)
     returns H$\{7+\hat\alpha_3\}$:H$\{7+\hat\alpha_4\}^{t_5\,[\text{(c)}\ \hat\alpha_{15}=\hat\alpha_3=\hat\alpha_4=0]}$ {
    t_3 G4:G$\{6+\hat\alpha_5\}^{[\hat\alpha_5 \geq -3]}$

[20] One might ask, why did we add this delta variable only to immediately constrain it to zero? Ensuring that every tile has a bottom-row delta variable makes the constraint generation for references slightly easier to state, e.g., in step 1 in Section 7.3.

$$\begin{array}{ll} \text{Intermediate subst} & \theta \ ::= \ \varepsilon \ \mid \ \theta, \hat{\alpha} \mapsto \hat{\beta} \ \mid \ \theta, \hat{\alpha} \mapsto 0 \\ \text{Delta substitution} & \Theta \ ::= \ \varepsilon \ \mid \ \Theta, \hat{\alpha} \mapsto 0 \ \mid \ \Theta, \hat{\alpha} \mapsto \alpha - l \end{array}$$

Fig. 6: Substitutions

$= \mathsf{F}\{4+\hat{\alpha}_6\}:\mathsf{F}\{4+\hat{\alpha}_7\}^{t_1}[\text{(a) } \hat{\alpha}_1=\hat{\alpha}_5, \hat{\alpha}_6=\hat{\alpha}_7=0]$

$\quad * \$\mathsf{G}\$\{2+\hat{\alpha}_8\}:\$\mathsf{G}\$\{2+\hat{\alpha}_9\}^{t_2}[\text{(c) } \hat{\alpha}_2=\hat{\alpha}_8=\hat{\alpha}_9=0]$

$t_4 \ \mathsf{H4}:\mathsf{H}\{6+\hat{\alpha}_{10}\}^{[\hat{\alpha}_{10}\geq -3]}$

$= \mathsf{F}\{4+\hat{\alpha}_{11}\}:\mathsf{F}\{4+\hat{\alpha}_{12}\}^{t_1}[\text{(a) } \hat{\alpha}_1=\hat{\alpha}_{10}, \hat{\alpha}_{11}=\hat{\alpha}_{12}=0]$

$\quad + \mathsf{G}\{4+\hat{\alpha}_{13}\}:\mathsf{G}\{4+\hat{\alpha}_{14}\}^{t_3}[\text{(a) } \hat{\alpha}_5=\hat{\alpha}_{10}, \hat{\alpha}_{13}=\hat{\alpha}_{14}=0]$

$t_5 \ \mathsf{H7}:\mathsf{H}\{7+\hat{\alpha}_{15}\}^{[\hat{\alpha}_{15}=0]}$

$= \mathsf{SUM}(\mathsf{H}\{4+\hat{\alpha}_{16}\}:\mathsf{H}\{6+\hat{\alpha}_{17}\}^{t_4}[\text{(b) } \hat{\alpha}_{16}=0, \hat{\alpha}_{17}=\hat{\alpha}_{10}]) \ \}$

At the definition of t_3, we see the constraint $\hat{\alpha}_5 \geq -3$, which keeps the height of the tile from becoming negative. On the other hand, the definition of t_5 has the constraint $\hat{\alpha}_{15} = 0$ because t_5 has height 1 in the original SDF, so it is not allowed to resize in the elastic SDF. We have not yet given the constraint generation algorithm for range references, but we can still understand the purpose of the constraints intuitively. For the reference $\mathsf{F}\{4+\hat{\alpha}_6\}:\mathsf{F}\{4+\hat{\alpha}_7\}^{t_1}$ appearing in t_3, the constraint $\hat{\alpha}_1 = \hat{\alpha}_5$ ensures that t_3 resizes along with t_1 according to apparent user intent. The constraint $\hat{\alpha}_6 = \hat{\alpha}_7 = 0$ ensures that the reference continues to point to t_1, subject only to copy/paste, and does not move due to the addition of extraneous variables. For the reference $\mathsf{H}\{4+\hat{\alpha}_{16}\}:\mathsf{H}\{6+\hat{\alpha}_{17}\}^{t_4}$ in t_5, which originally covered the entirety of t_4, the constraint $\hat{\alpha}_{16} = 0$ ensures that the top of the reference continues to point to the top of t_4, while $\hat{\alpha}_{17} = \hat{\alpha}_{10}$ ensures that the bottom of the reference moves with the bottom of t_4.

3. Solving the constraints yields the delta substitution that maps $\hat{\alpha}_1$, $\hat{\alpha}_5$, $\hat{\alpha}_{10}$, and $\hat{\alpha}_{17}$ to $\alpha - 3$ and all other delta variables to zero.
4. The result of applying this substitution to $\overline{\mathscr{F}}_0$ is the elastic SDF shown in Section 4.1 (Step 2).

7.2 Constraint solving

The business of the constraint solver is to find the principal (i.e. most general, up to renaming) delta substitution that satisfies the constraints, eliminating delta variables $\hat{\alpha}$ in favour of non-negative length variables α. A delta substitution Θ *satisfies* a constraint if applying Θ to both sides of the constraint makes the constraint true for all non-negative values of length variables. For example $\hat{\alpha} \mapsto \alpha + 3$ satisfies $\hat{\alpha} \geq 3$ because applying the substitution to both sides gives $\alpha + 3 \geq 3$, which is true for all $\alpha \geq 0$.

A delta substitution Θ_1 is *more general than* Θ_2 iff there is a length substitution ϕ such that $\Theta_2 = \phi \circ \Theta_1$. It is easy to compute the most general solution of a set of constraints:

1. *Eliminate all the equality constraints*. Gather all the equality constraints $\hat{\alpha} = \hat{\beta}$ and $\hat{\alpha} = 0$ into an *intermediate substitution* θ (Figure 6). That leaves only lower-bound constraints.

2. *Simplify lower bounds.* Apply θ to the lower bound constraints and combine all the constraints on each individual variable by taking the maximum of its lower bounds. Now we have a single constraint $\hat{\alpha} \geq -l$ for each delta variable $\hat{\alpha}$.
3. *Replace delta variables with length variables.* For each constraint $\hat{\alpha} \geq -l$, invent a fresh length variable α and compose θ with $\hat{\alpha} \mapsto \alpha - l$.

We will show later that every delta variable is transitively constrained equal to zero or to a delta variable with a lower bound, and hence the third step eliminates all delta variables in favour of length variables. Hence the result is a delta substitution Θ^*, with only length variables in its range (Figure 6).

Proposition 2. *Let Θ^* be the delta substitution produced by the constraint solver. Then it is principal in the sense that for every delta substitution Θ' satisfying the constraints, there exists a length substitution ϕ such that $\Theta' = \phi \circ \Theta^*$.*

Proof The only way Θ' can satisfy an equality constraint is if both sides of the constraint map to the same expression under Θ'. Every delta variable $\hat{\alpha}$ for which $\Theta^*(\hat{\alpha}) = 0$ was transitively constrained equal to 0, so $\Theta'(\hat{\alpha})$ must be 0 as well. Thus we will have $\Theta'(\hat{\alpha}) = (\phi \circ \Theta^*)(\hat{\alpha})$ regardless of ϕ.

The remaining delta variables fall into groups, where the members of each group were transitively constrained equal. For each group, the constraint solver picks a representative delta variable $\hat{\alpha}$, finds the maximum lower bound $-l$ that applied to any variable in the group (call this variable $\hat{\beta}$), and maps all members of the group to $\alpha - l$ where α is a fresh length variable. If $\Theta'(\hat{\alpha}) = 0$, then we simply choose $\phi(\alpha) = l$. If $\Theta'(\hat{\alpha})$ is of the form $\gamma - m$, then we must have $m \leq l$. (Otherwise, we would have $\Theta'(\hat{\beta}) = \gamma - m$ because of equality constraints, and then setting $\gamma = 0$ would violate the lower bound constraint $\hat{\beta} \geq -l$.) In this case, we choose $\phi(\alpha) = \gamma + l - m$. Either way, we have ensured that $\Theta'(\hat{\alpha}) = (\phi \circ \Theta^*)(\hat{\alpha})$ for all $\hat{\alpha}$ in the group. This completes the proof that $\Theta' = \phi \circ \Theta^*$, as desired. ■

7.3 Constraint generation for references

As discussed in Step 2 of Section 7.1, for each reference in $\overline{\mathscr{F}}_0$ we will generate some constraints. These constraints ensure that the corresponding reference in the generalisation $\overline{\mathscr{F}}$ is well-behaved, in the sense of Section 6.9. If we generate too few constraints, $\overline{\mathscr{F}}^*$ may contain ill-behaved references, and hence fail to be a *regular* generalisation of $\mathscr{F}$. But we must be careful: if we generate too many constraints, we may rule out some regular generalisations of $\mathscr{F}$, so that $\overline{\mathscr{F}}^*$ would fail to be the *principal* regular generalisation.

The constraints we generate must therefore express well-behavedness, using the lock-step/whole/inelastic reference kinds described in Section 6.5 and formalised in Section 6.9. For each range reference $\overline{\rho}$ in $\overline{\mathscr{F}}_0$, corresponding to a range reference ρ in $\mathscr{F}$, we generate constraints as follows:

1. Let t_t and t_c be the target tile and calling tile of $\overline{\rho}$, let $\overline{\sigma} = \mu_1\{m_1 + \hat{\alpha}_1\} : \mu_2\{m_2 + \hat{\alpha}_2\}$ be its row span reference, and let $\hat{\alpha}_t$ and $\hat{\alpha}_c$ be the bottom-row delta variables of t_t and t_c. Try the following cases in order:

a. *(Lockstep)* If $m_1 = m_2$, $\mu_1 = \mu_2$ is relative, and t_t and t_c have the same height in $\mathscr{F}$ (at least 2), then we anticipate $\overline{\sigma}$ is Lockstep, but it may turn out to be Inelastic. Constrain $\hat{\alpha}_t = \hat{\alpha}_c$ and $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$. (In this case, m_1 must point to the top row of t_t in $\mathscr{F}$ in order for ρ to stay within its single target tile t_t when copy/pasted through a calling tile of the same height.)

b. *(Whole)* Otherwise, if all of the following conditions hold:

   * The height of t_t in $\mathscr{F}$ is at least 2.
   * μ_1 and μ_2 are both absolute or the height of t_c in $\mathscr{F}$ is 1.
   * m_1 is the top row of t_t in $\mathscr{F}$.
   * m_2 is the bottom row of t_t in $\mathscr{F}$.

   Then we anticipate $\overline{\sigma}$ is Whole, but it may turn out to be Inelastic. Constrain $\hat{\alpha}_1 = 0$ and $\hat{\alpha}_2 = \hat{\alpha}_t$.

c. *(Inelastic)* Otherwise, $\overline{\sigma}$ is Inelastic. Constrain $\hat{\alpha}_t = \hat{\alpha}_1 = \hat{\alpha}_2 = 0$. In addition, if either μ_1 or μ_2 is relative, then constrain $\hat{\alpha}_c = 0$.

2. Follow the analogue of step (1) for the column axis.

An example of the output of this procedure was given in Section 7.1 (step 2).

7.4 Correctness of the inference algorithm

In this section we prove that our elasticity inference algorithm indeed finds the principal regular generalisation of a SDF $\mathscr{F}$; in doing so, we prove that a principal regular generalisation exists at all.

First we show that every delta variable is transitively constrained equal to zero or to a delta variable with a lower bound, as required by the constraint solver (Section 7.2). Indeed, every tile bottom-row and right-column delta variable is directly constrained to zero or given a lower bound in step 2a of the main algorithm. Each of the three cases of the reference constraint generation algorithm (Section 7.3) constrains the delta variables $\hat{\alpha}_1$ and $\hat{\alpha}_2$ of a row span reference equal to either zero or a tile delta variable (which, as we have just argued, meets the requirement of the constraint solver), and the analogous procedure applies to column span references. Thus the requirement is satisfied.

Next we deal with the most complex part of the proof: proving that the reference constraints properly characterise the well-behavedness of references. I.e., for a range reference $\overline{\rho}$ in $\overline{\mathscr{F}}_0$ and a delta substitution Θ, $\Theta(\overline{\rho})$ should be well-behaved if and only if Θ satisfies the constraints generated for $\overline{\rho}$. However, we will find that this equivalence holds only if $\Theta(\overline{\mathscr{F}}_0)$ already satisfies all the conditions to be a regular generalisation other than reference well-behavedness; this is OK because we will be able to establish those other conditions first. Thus we define:

Definition 7. *An elastic SDF $\overline{\mathscr{F}}$ is a* semi-regular generalisation *of $\mathscr{F}$ if it satisfies conditions (1)–(3) of Definition 4, i.e., all conditions for a regular generalisation except well-behavedness of references.*

We can now prove the main lemma about well-behaved references:

Lemma 1. *Let Θ be a delta substitution such that $\Theta(\overline{\mathscr{F}}_0)$ is an elastic SDF*[21] *$\overline{\mathscr{F}}$ that is a semi-regular generalisation of $\mathscr{F}$. Let $\overline{\rho}_0$ be a reference in $\overline{\mathscr{F}}_0$, and let $\overline{\rho} = \Theta(\overline{\rho}_0)$ be the corresponding reference in $\Theta(\overline{\mathscr{F}}_0)$. Then Θ satisfies the constraints generated for $\overline{\rho}_0$ if and only if $\overline{\rho}$ is well-behaved.*

Proof Both well-behavedness and constraint generation are independent for rows and columns, so it suffices to consider the row span references $\overline{\sigma}_0$ of $\overline{\rho}_0$ and $\overline{\sigma}$ of $\overline{\rho}$ and prove that Θ satisfies the constraints generated for $\overline{\sigma}_0$ if and only if $\overline{\sigma}$ is well-behaved.

Let $\sigma = \mu_1 m_1 : \mu_2 m_2$ be the row span reference of the original reference in $\mathscr{F}$. Let t_t, t_c, $\hat{\alpha}_t$, $\hat{\alpha}_c$, $\hat{\alpha}_1$, and $\hat{\alpha}_2$ be defined as in the constraint generator. By construction of $\overline{\mathscr{F}}_0$, we have $\overline{\sigma}_0 = \mu_1\{m_1 + \hat{\alpha}_1\} : \mu_2\{m_2 + \hat{\alpha}_2\}$. Then $\overline{\sigma} = \Theta(\overline{\sigma}_0) = \mu_1\{m_1 + \Theta(\hat{\alpha}_1)\} : \mu_2\{m_2 + \Theta(\hat{\alpha}_2)\}$. For convenience, let $\overline{m}_1 = m_1 + \Theta(\hat{\alpha}_1)$ and $\overline{m}_2 = m_2 + \Theta(\hat{\alpha}_2)$ so that $\overline{\sigma} = \mu_1 \overline{m}_1 : \mu_2 \overline{m}_2$.

To reduce duplicate reasoning, we first prove a few sub-lemmas.

Sub-lemma 1. *If $\overline{\sigma}$ is well-behaved of the Inelastic kind, then the constraints are satisfied no matter which case the constraint generator uses for $\overline{\sigma}_0$.*

Proof If $\overline{\sigma}$ is Inelastic, then we know that $\hat{\alpha}_t = \hat{\alpha}_1 = \hat{\alpha}_2 = 0$, and furthermore $\hat{\alpha}_c = 0$ if μ_1 or μ_2 is relative. It is clear that these statements imply the satisfaction of the constraints for each case of the constraint generator. (Notice that the Lockstep case can only be reached if μ_1 and μ_2 are relative, which ensures we have $\hat{\alpha}_c = 0$ in that case.) ■

Sub-lemma 2. *If $\overline{\sigma}$ is well-behaved of the Lockstep kind, then the constraint generator will use the Lockstep case for $\overline{\sigma}_0$.*

Proof If $\overline{\sigma}$ is Lockstep, then t_c and t_t have the same non-constant height in $\overline{\mathscr{F}}$. Since $\overline{\mathscr{F}}$ is a semi-regular generalisation of $\mathscr{F}$, their common height in $\mathscr{F}$ must be at least 2. We also know $\overline{m}_1 = \overline{m}_2$, so $m_1 = m_2$. Finally, we know μ_1 and μ_2 are relative. Thus, all the conditions are met for the constraint generator to use the Lockstep case. ■

Sub-lemma 3. *If $\overline{\sigma}$ is well-behaved of the Whole kind, then the constraint generator will use the Whole case.*

Proof If $\overline{\sigma}$ is Whole, then t_t has variable height in $\overline{\mathscr{F}}$. Again, since $\overline{\mathscr{F}}$ is a semi-regular generalisation of $\mathscr{F}$, the height of t_t in $\mathscr{F}$ must be at least 2. Since $\overline{m}_1$ points to the top row of t_t for all values of length variables, if we instantiate the length variables to recover $\mathscr{F}$, we find that m_1 is the top row of t_t in $\mathscr{F}$; likewise, m_2 is the bottom row of t_t in $\mathscr{F}$. Finally, we know that the height of t_c is 1 in $\overline{\mathscr{F}}$ (hence also in $\mathscr{F}$) or both μ_1 and μ_2 are absolute. Thus, all the conditions are met for the constraint generator to use the Whole case. We just have to show it does not use the Lockstep case first. That is clear, because the Lockstep case would require t_c to have height at least 2 in $\mathscr{F}$ and μ_1 and μ_2 to both be relative, which contradicts what we know. ■

[21] $\Theta(\overline{\mathscr{F}}_0)$ might not be an elastic SDF as we have defined it if it lacks the property that all tiles have nonnegative height and width for all values of length variables.

Now we break the proof into cases based on which case of the constraint generator is reached, and we prove the bidirectional implication between constraint satisfaction and well-behavedness in each case.

- *Lockstep constraint generation:* We know that t_c and t_t have the same height in $\mathscr{F}$ (which is at least 2), $m_1 = m_2$, $\mu_1 = \mu_2$ is relative, and m_1 points to the top row of t_t. The constraints are $\hat{\alpha}_t = \hat{\alpha}_c$ and $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$.

  - If the constraints are satisfied and t_t has *variable* height, then the heights of t_c and t_t in $\overline{\mathscr{F}}'$ are the same non-constant expression (since the same delta is added to the same starting height), $\overline{m}_1 = \overline{m}_2 = m_1 = m_2$, and hence $\overline{m}_1$ points to the top row of t_t. Thus $\overline{\sigma}$ indeed satisfies the conditions of the Lockstep kind.
  - If the constraints are satisfied and t_t has *constant* height, then our constraint $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$ ensures that $\overline{m}_1$ and $\overline{m}_2$ are constant. Also, our constraint $\hat{\alpha}_t = \hat{\alpha}_c$ ensures that t_c has constant height (in fact, the same as t_t). Thus $\overline{\sigma}$ satisfies the requirements of the Inelastic kind.
  - If $\overline{\sigma}$ is well-behaved, it cannot be Whole by Sub-lemma 3. If it is Inelastic, the constraints are satisfied by Sub-lemma 1. So suppose $\overline{\sigma}$ is Lockstep. Then the heights of t_t and t_c are the same non-constant expression; after we subtract the equal heights of t_t and t_c in $\mathscr{F}$, $\hat{\alpha}_t = \hat{\alpha}_c$ must be satisfied. Furthermore, $\overline{m}_1 = \overline{m}_2$ points to the top row of t_t, as does m_1, so $\overline{m}_1 = m_1$. Thus $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$ is satisfied.
- *Whole constraint generation:* We know that m_1 is the top row of t_t in $\mathscr{F}$ and m_2 is the bottom row of t_t in $\mathscr{F}$. We also know that μ_1 and μ_2 are both absolute or the height of t_c in $\mathscr{F}$ is 1; in the latter case, the height of t_c is still 1 in $\overline{\mathscr{F}}$ because $\overline{\mathscr{F}}$ is a semi-regular generalisation of $\mathscr{F}$. The constraints are $\hat{\alpha}_1 = 0$ and $\hat{\alpha}_2 = \hat{\alpha}_t$.

  - If the constraints are satisfied and t_t has *variable* height, then our constraints $\hat{\alpha}_1 = 0$ and $\hat{\alpha}_2 = \hat{\alpha}_t$ ensure that $\overline{m}_1$ and $\overline{m}_2$ point to the top and bottom of t_t in $\overline{\mathscr{F}}$. With the condition on μ_1, μ_2, and t_c, $\overline{\sigma}$ satisfies the requirements of the Whole kind.
  - If the constraints are satisfied and t_t has *constant* height, then $\Theta(\hat{\alpha}_t)$ must be 0, so our constraints $\hat{\alpha}_1 = 0$ and $\hat{\alpha}_2 = \hat{\alpha}_t$ ensure that $\overline{m}_1$ and $\overline{m}_2$ are constant. Also, we know that μ_1 and μ_2 are both absolute or the height of t_c is constant (in fact, 1), by the same logic as in the previous case. Thus $\overline{\sigma}$ satisfies the requirements of the Inelastic kind.
  - If $\overline{\sigma}$ is well-behaved, it cannot be Lockstep by Sub-lemma 2. If it is Inelastic, the constraints are satisfied by Sub-lemma 1. So suppose $\overline{\sigma}$ is Whole. Then $\overline{m}_1$ points to the top row of t_t and $\overline{m}_2$ points to the bottom row of t_t in $\overline{\mathscr{F}}$. It follows that $\hat{\alpha}_1 = 0$ and $\hat{\alpha}_2 = \hat{\alpha}_t$ are satisfied.
- *Inelastic constraint generation:* The constraints are $\hat{\alpha}_t = \hat{\alpha}_1 = \hat{\alpha}_2 = 0$. If μ_1 or μ_2 is relative, there is a further constraint $\hat{\alpha}_c = 0$.

  - If the constraints are satisfied, then t_t has constant height and $\overline{m}_1$ and $\overline{m}_2$ are constant. Furthermore, either μ_1 and μ_2 are both absolute or our constraint $\hat{\alpha}_c = 0$ ensures that t_c has constant height. Thus, $\overline{\sigma}$ satisfies the requirements of the Inelastic kind.

– If $\overline{\sigma}$ is well-behaved, it cannot be of the Lockstep or Whole kind by Sub-lemmas 2 and 3. So it must be Inelastic, and it satisfies the constraints by Sub-lemma 1.

■

Finally, we proceed to the key theorem:

Theorem 2. *Every basic SDF $\mathscr{F}$ has a principal regular generalisation $\overline{\mathscr{F}}^*$, and the algorithm in Section 7.1 finds it.*

Proof Let $\overline{\mathscr{F}}^* = \Theta^*(\overline{\mathscr{F}}_0)$ be the elastic SDF found by the algorithm. (There is a nit: to be an elastic SDF as we have defined it, $\overline{\mathscr{F}}^*$ must have nonnegative tile heights and widths, but this is guaranteed by the tile size constraints generated in step 2a of the main algorithm.) We need to show that (A) $\overline{\mathscr{F}}^*$ is a regular generalisation of $\mathscr{F}$ and (B) it is more general than every other regular generalisation of $\mathscr{F}$.

(A): $\overline{\mathscr{F}}^*$ satisfies condition 2 for regularity because we constructed $\overline{\mathscr{F}}_0$ with constant tile upper-left corners. It satisfies condition 3 because of the tile size constraints generated in step 2a of the main algorithm. Next we claim $\overline{\mathscr{F}}^*$ is a generalisation of $\mathscr{F}$ (condition 1 for regularity). Consider the structure of Θ^* as produced by the constraint solver: some delta variables map to zero and others map to expressions of the form $\alpha - l$. The same length variable α may appear several times, but always in combination with the same l. Thus, we can construct a length substitution ϕ that maps each length variable α to the corresponding l. Then $\phi \circ \Theta^*$ maps all delta variables to zero, so $(\phi \circ \Theta^*)(\overline{\mathscr{F}}_0) = \mathscr{F}$. Thus $\phi(\overline{\mathscr{F}}^*) = \mathscr{F}$, which witnesses that $\overline{\mathscr{F}}^*$ is a generalisation of $\mathscr{F}$.

At this point, we have proved that $\overline{\mathscr{F}}^*$ satisfies conditions 1–3 to be a regular generalisation of $\mathscr{F}$, i.e., it is a semi-regular generalisation of $\mathscr{F}$. Thus, we can apply Lemma 1: since Θ^* satisfies the constraints generated for every reference in $\overline{\mathscr{F}}_0$, every reference in $\overline{\mathscr{F}}^*$ is well-behaved. Therefore, $\overline{\mathscr{F}}^*$ is a regular generalisation of $\mathscr{F}$.

(B): Let $\overline{\mathscr{F}}$ be a regular generalisation of $\mathscr{F}$. First we claim there is a delta substitution Θ such that $\overline{\mathscr{F}} = \Theta(\overline{\mathscr{F}}_0)$. Since $\overline{\mathscr{F}}$ is a generalisation of $\mathscr{F}$, there is a length substitution ϕ such that $\mathscr{F} = \phi(\overline{\mathscr{F}})$. Since length substitutions only affect length variables appearing in elastic axis positions, the only places $\overline{\mathscr{F}}$ can differ from $\mathscr{F}$ are elastic axis positions in tile ranges and range references. By regularity condition 2, the upper-left corners of tiles in $\overline{\mathscr{F}}$ must be constant. In every other place, $\overline{\mathscr{F}}$ has an elastic axis position $\overline{m}$ of the form m or $m + \alpha$, while $\overline{\mathscr{F}}_0$ has an expression $m_0 + \hat{\alpha}_i$ for a fresh delta variable $\hat{\alpha}_i$, where m_0 was the position in $\mathscr{F}$. If $\overline{m} = m$, then we must have $m = m_0$, otherwise $\overline{\mathscr{F}}$ could not be specialised to $\mathscr{F}$; in this case, we add $\hat{\alpha}_i \mapsto 0$ to Θ. If $\overline{m} = m + \alpha$, then we must have $m \leq m_0$, otherwise $\overline{\mathscr{F}}$ could not be specialised to $\mathscr{F}$ (because α must be nonnegative); in this case, we add $\hat{\alpha}_i \mapsto \alpha + m - m_0$ to Θ. (Note that $m - m_0 \leq 0$ as required for a delta substitution.) In either case, Θ maps the expression in $\overline{\mathscr{F}}_0$ to the one in $\overline{\mathscr{F}}$. Thus, we have constructed the required Θ.

Since $\overline{\mathscr{F}}$ satisfies condition 3 for regularity, it is easy to show that Θ satisfies the tile size constraints from step 2a of the main algorithm. Furthermore, since $\overline{\mathscr{F}}$ is a semi-regular (in fact, regular) generalisation of $\mathscr{F}$ in which all references are well-behaved, by Lemma 1, Θ satisfies the constraints generated for every reference. In sum, Θ satisfies all the constraints. By Proposition 2, there is a length substitution ϕ such that $\Theta = \phi \circ \Theta^*$. Then we have

$\phi(\overline{\mathscr{F}}^*) = \phi(\Theta^*(\overline{\mathscr{F}}_0)) = \Theta(\overline{\mathscr{F}}_0) = \overline{\mathscr{F}}$, so $\overline{\mathscr{F}}^*$ is more general than $\overline{\mathscr{F}}$. This completes the proof. ■

It is worth emphasizing: *every* basic SDF $\mathscr{F}$ has a principal regular generalisation $\overline{\mathscr{F}}^*$ and the algorithm finds it. If $\mathscr{F}$ uses a computational pattern that our system cannot handle, it is not an error; what normally happens is that $\overline{\mathscr{F}}^*$ is less general than the user envisioned.

While it is not essential to the preceding point, the following property is interesting and should come as no surprise:

Proposition 3. *Every basic SDF $\mathscr{F}$ is a regular generalisation of itself.*

Proof It's clear that the delta substitution Θ that maps every delta variable to 0 satisfies all constraints of the allowed forms. By reasoning similar to part (A) of Theorem 2, $\Theta(\overline{\mathscr{F}}_0)$ satisfies all the conditions to be a regular generalisation of $\mathscr{F}$. But $\Theta(\overline{\mathscr{F}}_0) = \mathscr{F}$. ■

So in the worst case, if $\mathscr{F}$ does not use any computational pattern that we can generalise, we get $\overline{\mathscr{F}}^* = \mathscr{F}$.

As mentioned before, $\overline{\mathscr{F}}^*$ may fail to be determinable, in which case we recommend setting all non-determinable length variables to their initial values.

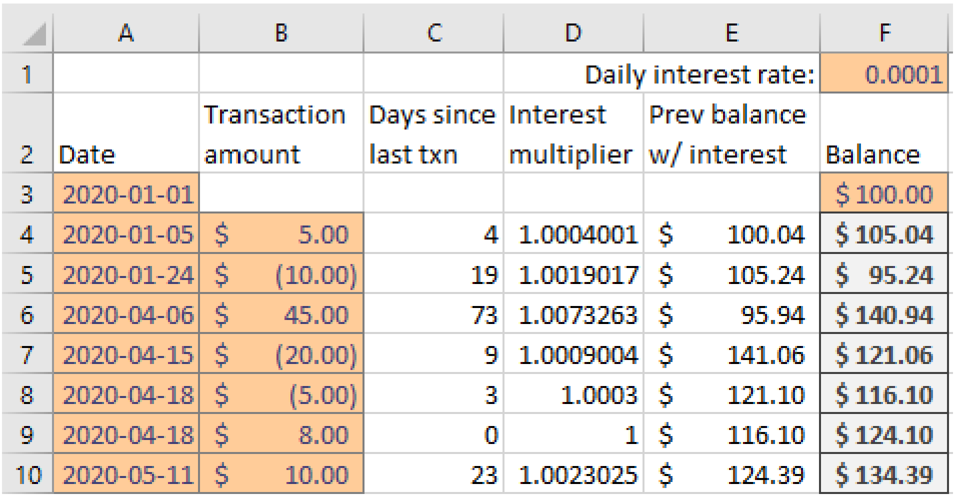
8 An extended generalisation system

In this section, we describe an extension of the system of Sections 6–7 to support more interesting SDFs, including *non-basic* SDFs; that is, SDFs that have range references with multiple target tiles. The only parts of the system we will have to change are the definition of well-behaved references (which is where the restriction to basic SDFs previously came from) and the related constraint generation and proofs.

The extra expressiveness of the new system is important in practice. As a motivating example, below is a function that computes the post-transaction balances of a bank account with interest compounded daily. The sheet, with example data but without formulas visible, is in Figure 7.

```
/* COMPOUND( start date, opening balance, interest rate, transactions )
   transactions is a 2-column array of (date, amount) pairs */
function COMPOUND( $t_s$ A3, $t_o$ F3, $t_r$ F1, $t_x$ A4:B10 ) returns F4:F10$^{t_4}$ {
  $t_1$ C4:C10 = A4$^{t_x}$ − A3$^{t_s,t_x}$              /* Interval between transactions */
  $t_2$ D4:D10 = POWER(1+$F$1$^{t_r}$, C4$^{t_1}$)    /* Interest multiplier */
  $t_3$ E4:E10 = F3$^{t_o,t_4}$ * D4$^{t_2}$              /* New balance after interest */
  $t_4$ F4:F10 = E4$^{t_3}$ + B4$^{t_x}$  }               /* Final balance */
```

The function takes (as its last argument) a 2-column array of transactions. It computes each new balance by adding a suitable interest payment (which depends on the date interval) and the transaction amount, and returns an array of the post-transaction balances. We have carefully placed the start date in A3 immediately above the column of transaction dates in A4:A10, so that we can uniformly compute the intervals during which interest accrues. Now consider the reference to A3 in the definition of t_1. When resolved at cell C4, the reference A3 points to the start date input tile t_s; but when resolved at cell C5, the A3 has become A4 (via copy/paste), and hence points to the date on the first transaction, in tile t_x.



	A	B	C	D	E	F
1					Daily interest rate:	0.0001
2	Date	Transaction amount	Days since last txn	Interest multiplier	Prev balance w/ interest	Balance
3	2020-01-01					$ 100.00
4	2020-01-05	$ 5.00	4	1.0004001	$ 100.04	$ 105.04
5	2020-01-24	$ (10.00)	19	1.0019017	$ 105.24	$ 95.24
6	2020-04-06	$ 45.00	73	1.0073263	$ 95.94	$ 140.94
7	2020-04-15	$ (20.00)	9	1.0009004	$ 141.06	$ 121.06
8	2020-04-18	$ (5.00)	3	1.0003	$ 121.10	$ 116.10
9	2020-04-18	$ 8.00	0	1	$ 116.10	$ 124.10
10	2020-05-11	$ 10.00	23	1.0023025	$ 124.39	$ 134.39

Fig. 7: The COMPOUND spreadsheet to compute post-transaction balances of a bank account with interest compounded daily.

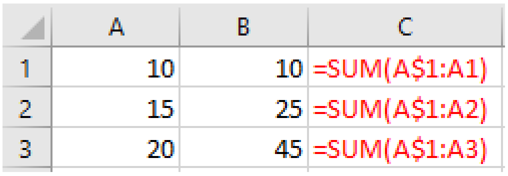
So this reference to A3 has *two* target tiles, t_s and t_x, and is therefore labelled with both, making it a non-basic SDF.

Fortunately, the new system handles arbitrary tame SDFs and still enjoys principal generalisations. For instance, our algorithm generalises the SDF above to the following elastic SDF:

```
function COMPOUND( $t_s$ A3, $t_o$ F3, $t_r$ F1, $t_x$ A4:B$\{3+\alpha\}$ ) returns F4:F$\{3+\alpha\}^{t_4}$ {
  $t_1$ C4:C$\{3+\alpha\}$ = A4$^{t_x}$ − A3$^{t_s,t_x}$
  $t_2$ D4:D$\{3+\alpha\}$ = POWER(1+$F$1$^{t_r}$, C4$^{t_1}$)
  $t_3$ E4:E$\{3+\alpha\}$ = F3$^{t_o,t_4}$ $*$ D4$^{t_2}$
  $t_4$ F4:F$\{3+\alpha\}$ = E4$^{t_3}$ + B4$^{t_x}$
}
```

8.1 Overview of the extended system

In general, our main goal is to allow relative references with an upward offset between tiles of the same height, where the "offset" refers to positions within the tiles, regardless of the locations of the tiles in the sheet. So for example, if $\overline{r}_1$ and $\overline{r}_2$ are tiles of height α and width 1 (or possibly the same tile) and $\overline{r}_1$ contains a relative reference to $\overline{r}_2$, then it may be the case that the second row of $\overline{r}_1$ points to the first row of $\overline{r}_2$, the third row of $\overline{r}_1$ to the second row of $\overline{r}_2$, and so forth. Of course, the first row of $\overline{r}_1$ needs somewhere to point; it must point to a separate, vertically inelastic tile located immediately above $\overline{r}_2$, like tiles t_s and t_o in the COMPOUND example. Analogous remarks apply to leftward offsets. We do not support downward and rightward offsets because supporting offsets in both directions on the same axis would complicate the algorithm for little benefit. Computations with downward and rightward offsets are rare outside recreational contexts (e.g., cellular automata), and in some cases they can be expressed in our system by using an absolute



	A	B	C
1	10	10	=SUM(A$1:A1)
2	15	25	=SUM(A$1:A2)
3	20	45	=SUM(A$1:A3)

Fig. 8: The CUMSUM spreadsheet demonstrating cumulative aggregation.

reference to the entire target tile and then using the INDEX function to extract the desired element.

Regularity condition 3, which requires that the upper-left corner of each tile be constant, makes it possible to position an inelastic tile immediately above or to the left of an elastic tile to fulfill a reference with an upward or leftward offset but does not make it possible to position an inelastic tile below or to the right of an elastic tile to support a downward or rightward offset, because in the latter case, the upper-left corner of the inelastic tile would depend on the size of the elastic tile.

In addition to adding support for upward and leftward offsets (which goes hand in hand with allowing references with multiple target tiles), we add support for cumulative aggregation. For example, we would like to generalise:

```
function CUMSUM($t_1$ A1:A3) returns B1:B3$^{t_2}$ {
  $t_2$ B1:B3 = SUM(A$1:A1$^{t_1}$) }
```

(see the sheet in Figure 8) to:

```
function CUMSUM($t_1$ A1:A$\{\alpha\}$) returns B1:B$\{\alpha\}^{t_2}$ {
  $t_2$ B1:B$\{\alpha\}$ = SUM(A$1:A1$^{t_1}$) }
```

To do so, we move beyond the three kinds of span references in the simplified system and allow each endpoint of a span reference to independently fall into one of four kinds: Inelastic, Lockstep, Start, and End. The row span $1:1 in the reference A$1:A1 in the formula for t_2 has one Start endpoint $1 and one Lockstep endpoint 1.

In the original system, the only kind of row span reference that could contain a relative row reference from a calling tile of variable height was Lockstep, and it required that the heights of the calling and target tiles be exactly the same expression. How much more flexible can our Lockstep kind be and still ensure that every tame SDF has a principal regular generalisation? For example, consider an SDF that generates a list iteratively and has a special formula for the first element:

```
function SUM2($t_1$ A1:A3) returns B3$^{t_3}$ {
  $t_2$ B1 = A1$^{t_1}$
  $t_3$ B2:B3 = B1$^{t_2,t_3}$ + A2$^{t_1}$
}
```

(Of course, a better design in this case would be to initialise the sum to 0 and then use a consistent formula, but it's unclear to us whether such a transformation will always be natural to users in more complex cases.) We'd like the following generalisation:

```
function SUM2($t_1$ A1:A$\{1+\alpha\}$) returns B$\{1+\alpha\}^{t_3}$ {
  $t_2$ B1 = A1$^{t_1}$
  $t_3$ B2:B{1+$\alpha$} = B1$^{t_2,t_3}$ + A2$^{t_1}$
}
```

in which t_3 has variable height and contains a relative reference to t_1, even though their heights differ by 1. We find that in general, we can allow the heights of the caller and target tiles to differ by a constant and still have a principal regular generalisation. The principle that we allow upward but not downward offsets still holds if we consider positions with respect to the *bottoms* of the tiles: the last row of the caller tile may point at or above the last row of the target tile, but not below because there is no way to position another target tile there. As always, analogous remarks apply to columns.

8.2 Well-behaved references

We can now proceed to the definition of a well behaved reference, replacing the one in Section 6.9. First we need some auxiliary definitions to deal with the multiple target tiles. The *overall target range* of a reference ρ in the original SDF $\mathscr{F}$ is the range of all cells read by ρ resolved at any cell of the calling tile. If $\rho = \theta_1 : \theta_2$ and the range of the calling tile is $a_1 : a_2$, then the overall target range can be computed as $\mathrm{Res}(\theta_1, a_1) : \mathrm{Res}(\theta_2, a_2)$.[22] A target tile of ρ is *vertically final* if it overlaps the last row of the overall target range and *horizontally final* if it overlaps the last column. When we are discussing an elastic SDF $\overline{\mathscr{F}}$ as a potential generalisation of $\mathscr{F}$, a target tile of a reference $\overline{\rho}$ is said to be vertically or horizontally final if the corresponding target tile of the corresponding reference in $\mathscr{F}$ is. (We make this definition because the layout of the target tiles in $\mathscr{F}$ provides a template that we want to use to analyse the the layout of the target tiles in the elastic SDF.)

Definition 8 (Well behaved reference in the extended system)**.** *In an elastic SDF $\overline{\mathscr{F}}$, a range reference $\overline{\rho}$ with calling tile t_c is* well behaved *if it is well behaved on both axes. We give the definition for the row axis; the one for the column axis is analogous. Let $\overline{\sigma} = \mu_1\overline{m}_1 : \mu_2\overline{m}_2$ be the row span reference of $\overline{\rho}$. $\overline{\rho}$ is well behaved on the row axis if it satisfies the following conditions:*

1. *Every vertically non-final target tile has constant height.*
2. *Every vertically final target tile t_t stands in one of the following relationships to each of the two row references $\mu_i\overline{m}_i$ in $\overline{\sigma}$:*
   - Inelastic: *The height of t_t is constant, $\overline{m}_i$ is constant, and μ_i is absolute or the height of t_c is constant.*
   - Lockstep: *The heights of t_c and t_t are non-constant but their difference is a constant, $\overline{m}_i$ is constant, and μ_i is relative.*
   - Start: *The height of t_t is of the form $h_t + \alpha_t$, and μ_i is absolute or the height of t_c is 1. Furthermore, $\overline{m}_i$ points at a constant non-negative offset above the top row of t_t, and if this offset is zero and $i = 2$ (i.e., $\mu_i\overline{m}_i$ is the bottom endpoint of $\overline{\sigma}$), then $h_t > 0$.*

[22] All of these cells are actually read, since our assumption that $\mathscr{F}$ is non-degenerate rules out the possibility that ρ is a relative reference to a zero-size range that moves across the overall target range without reading any cells.

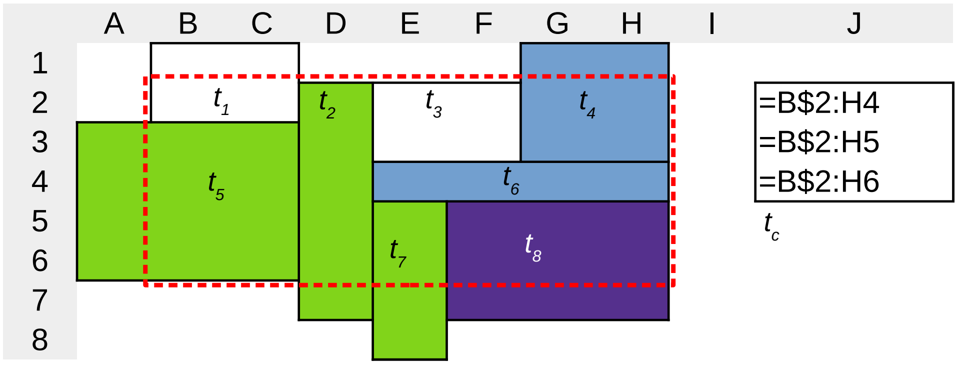


Fig. 9: A (crazy) example well-behaved reference in the extended system.

– End: *The height of t_t is of the form $h_t + \alpha_t$, and μ_i is absolute or the height of t_c is 1. Furthermore, $\overline{m}_i$ points to the bottom row of t_t, and if $i = 1$ (i.e., $\mu_i\overline{m}_i$ is the top endpoint of $\overline{\sigma}$), then $h_t > 0$.*

Furthermore, the pair of relationships of $\mu_1\overline{m}_1$ and $\mu_2\overline{m}_2$ to t_t must not be (Lockstep, Start) or (End, Lockstep).[23]

We give an example of a well-behaved reference that is crazier than anything that would be expected in practice but serves to illustrate what is possible under the definition. (The reader may imagine removing things from this example to get more realistic examples.) Figure 9 shows the fragment of the inelastic SDF, and the corresponding Calculation View is below:

t_c J2:J4 = SUM(B$2:H4^{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8})
t_1 B1:C2 = ...
t_2 D2:D7 = ...
t_3 E2:F3 = ...
t_4 G1:H3 = ...
t_5 A3:C6 = ...
t_6 E4:H4 = ...
t_7 E5:E8 = ...
t_8 F5:H7 = ...

The reference ρ has calling tile t_c with range J2:J4, and in the formula for the upper-left caller cell J2, it is written B$2:H4. The bottom row of ρ is relative and updates as expected when ρ is copied down to the remaining cells of t_c. So ρ, resolved at its various calling cells, retrieves arrays that are cumulative vertically but have a fixed horizontal span. The overall target range of ρ is B2:H6, marked by the red dotted rectangle. ρ has eight target tiles $t_1, \ldots, t_8$. t_5, t_2, t_7 are vertically final (green); t_4 and t_6 are horizontally final (light blue); and t_8 is both vertically and horizontally final (dark blue).

The principal regular generalisation of the sheet fragment is as follows:

t_c J2:J$\{1+\alpha\}$ = SUM(B$2:$\{F+\beta\}4^{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8}$)

[23] If these pairs of relationships occurred, then $\overline{\sigma}$ would resolve to a negative-height row span at most rows of t_c when t_c and t_t increase in height. If we did not forbid them here, these pairs could occur in a corner case if t_c and t_t both have initial height exactly 2, while (End, Start), which would cause a similar problem, is ruled out by the non-degeneracy condition.

t_1 B1:C2 = ...
t_2 D2:D{4+α} = ...
t_3 E2:F3 = ...
t_4 G1:{F+β}3 = ...
t_5 A3:C{3+α} = ...
t_6 E4:{F+β}4 = ...
t_7 E5:E{5+α} = ...
t_8 F5:{F+β}{4+α} = ...

As required, the vertically non-final tiles t_1, t_3, t_4, and t_6 have constant height, while the horizontally non-final tiles t_1, t_2, t_3, t_5, and t_7 have constant width. The row span reference of $\overline{\rho}$ is \$2:4. The top endpoint \$2 stands in the Start relationship to each of the vertically final target tiles since it is absolute, they have variable height and it points at or above their top rows. The bottom endpoint 4 stands in the Lockstep relationship to each of the vertically final target tiles since it is relative and a constant and the heights of the caller and vertically final target tiles all have the same length variable α, hence the height differences are constant.

The column span reference of $\overline{\rho}$ is B:{F+β}. The left endpoint B stands in the Start relationship to each of the horizontally final target tiles by similar reasoning as for the top endpoint, except it is not absolute, but that is OK because t_c has width 1. The right endpoint {F+β} stands in the End relationship to each of the horizontally final target tiles because they have variable width, it points to their rightmost columns, and (again) t_c has width 1.

As α increases, tiles t_5, t_2, t_7, and t_8 all grow downward, and t_c grows downward with additional copy/paste adjustments of $\overline{\rho}$ that refer to most of the new cells. α can also decrease as far as 0. When $\alpha = 1$, $\overline{\rho}$ no longer reads from t_7 and t_8; this can happen with Lockstep references. Similar remarks apply with β and horizontal resizing, except that the right edges of t_4, t_6, and t_8 are referenced by every cell of t_c. Since the only way target tiles grow is that vertically final tiles grow down and horizontally final tiles grow to the right, intuitively we can see that there is no way two target tiles of $\overline{\rho}$ can overlap; this will be proved in the next section. Finally, we note that if α were set to a constant by other constraints, both row references of $\overline{\rho}$ would stand in the Inelastic relationship to the vertically final target tiles; the analogous statement holds for β, the column references, and the horizontally final target tiles.

We give a second example to illustrate an important case that could not be incorporated into the first example: a row span reference with both endpoints in the End relationship to the vertically final target tiles. Figure 10 shows the fragment of the inelastic SDF, and here is the Calculation View:

t_c B7:D7 = B5:B5^{t_1,t_2}
t_1 A2:C5 = ...
t_2 D1:E5 = ...

Here is the well-behaved generalisation of interest (not principal, since to keep things simpler, we have declined to introduce horizontal elasticity):

t_c B7:D7 = B{2+α}:B{2+α}^{t_1,t_2}
t_1 A2:C{2+α} = ...
t_2 D1:E{2+α} = ...

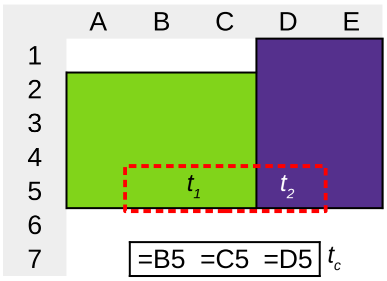


Fig. 10: A well-behaved reference with both endpoints of the row span reference in the End relationship.

Both row references are $\{2{+}\alpha\}$, and they stand in the End relationship to the target tiles. As α changes, t_1 and t_2 resize, and the reference continues to select the relevant cells from their last rows. If α decreases, the reference reads cells above the overall target range from the original SDF. This can only happen when both row references have the End relationship, because that is the only way the top row reference contains a length variable (which can decrease from the value that recovers the original SDF). If the top row reference has constant coordinate (whether absolute or relative), it is the same constant as in the inelastic SDF and serves as a lower bound on the row number of any cell that can be read by the range reference.

One further point: for the top row reference to have the End relationship to the target tiles, they must have height at least 1 for all values of length variables, so that they have a bottom row for the range reference to access. We can see this is true of t_1 and t_2 in the example. This requirement can sometimes pose a problem. For example, if we modify the COMPOUND function at the beginning of Section 8 to return just the final balance instead of the column of post-transaction balances:

```
/* COMPOUND2( start date, opening balance, interest rate, transactions )
   transactions is a 2-column array of (date, amount) pairs */
function COMPOUND2( $t_s$ A3, $t_o$ F3, $t_r$ F1, $t_x$ A4:B10 )
  returns F10$^{t_4}$ {
 $t_1$ C4:C10 = A4$^{t_x}$ − A3$^{t_s,t_x}$   /* Interval between transactions */
 $t_2$ D4:D10 = POWER(1+$F$1$^{t_r}$, C4$^{t_1}$)   /* Interest multiplier */
 $t_3$ E4:E10 = F3$^{t_o,t_4}$ ∗ D4$^{t_2}$   /* New balance after interest */
 $t_4$ F4:F10 = E4$^{t_3}$ + B4$^{t_x}$  }   /* Final balance */
```

we would get the following generalisation:

```
function COMPOUND2( $t_s$ A3, $t_o$ F3, $t_r$ F1, $t_x$ A4:B{4+$\alpha$} )
  returns F{4+$\alpha$}$^{t_4}$ {
 $t_1$ C4:C{4+$\alpha$} = A4$^{t_x}$ − A3$^{t_s,t_x}$
 $t_2$ D4:D{4+$\alpha$} = POWER(1+$F$1$^{t_r}$, C4$^{t_1}$)
 $t_3$ E4:E{4+$\alpha$} = F3$^{t_o,t_4}$ ∗ D4$^{t_2}$
 $t_4$ F4:F{4+$\alpha$} = E4$^{t_5}$ + B4$^{t_x}$
}
```

Because the top row reference of the returns reference had the End relationship to t_4, t_4 was required to have positive height, which propagated to t_x, t_1, t_2, and t_3. Now COMPOUND2 is unable to accept an array of zero transactions. Given such input, we would want it to return the opening balance. It may be possible to modify the system to allow the returns reference to read from t_o even though t_o was not in this reference's original target-tile set; we leave this question to future work. In the meantime, one workaround is to define an extra tile that copies both t_o and t_4 (which is allowed by the Lockstep relationship) and then reference the last cell of this tile:

```
function COMPOUND3( $t_s$ A3, $t_o$ F3, $t_r$ F1, $t_x$ A4:B10 )
  returns G10$^{t_5}$ {
 $t_1$ C4:C10 = A4$^{t_x}$ − A3$^{t_s,t_x}$   /* Interval between transactions */
 $t_2$ D4:D10 = POWER(1+$F$1$^{t_r}$, C4$^{t_1}$)   /* Interest multiplier */
 $t_3$ E4:E10 = F3$^{t_o,t_4}$ * D4$^{t_2}$   /* New balance after interest */
 $t_4$ F4:F10 = E4$^{t_3}$ + B4$^{t_x}$   /* Final balance */
 $t_5$ G3:G10 = F3$^{t_o,t_4}$ }
```

Now in the generalisation, t_x can shrink to zero height and t_5 will still have height 1, as required to reference its last cell:

```
function COMPOUND3( $t_s$ A3, $t_o$ F3, $t_r$ F1, $t_x$ A4:B{3+$\alpha$} )
  returns G{3+$\alpha$}$^{t_4}$ {
 $t_1$ C4:C{3+$\alpha$} = A4$^{t_x}$ − A3$^{t_s,t_x}$
 $t_2$ D4:D{3+$\alpha$} = POWER(1+$F$1$^{t_r}$, C4$^{t_1}$)
 $t_3$ E4:E{3+$\alpha$} = F3$^{t_o,t_4}$ * D4$^{t_2}$
 $t_4$ F4:F{3+$\alpha$} = E4$^{t_5}$ + B4$^{t_x}$
 $t_5$ G3:G{3+$\alpha$} = F3$^{t_o,t_4}$
}
```

8.3 Unambiguity of a regular generalisation

Following the earlier presentation (Section 6.10), we next prove that with our new definition of well-behaved references, every regular generalisation of an SDF is unambiguous.

Theorem 3. *Every regular generalisation $\overline{\mathscr{F}}$ of a tame SDF $\mathscr{F}$ is unambiguous.*

Proof We must show two conditions for each range reference $\overline{\rho}$ in $\overline{\mathscr{F}}$, which is well-behaved by our assumption that $\overline{\mathscr{F}}$ is regular. Let t_c be the calling tile of $\overline{\rho}$ and let $a_1 : \overline{a}_2$ be its range (by regularity, the top-left corner is constant). We need to show that for every length assignment ϕ, the target tiles of $\phi(\overline{\rho})$ do not overlap in $\phi(\overline{\mathscr{F}})$ and for every cell a in $\phi(a_1 : \overline{a}_2)$, the resolved target range $\text{Res}(\phi(\overline{\rho}), a)$ has non-negative height and width and is covered by the target tiles in $\phi(\overline{\mathscr{F}})$.

Let $\overline{\rho} = (\overline{\theta}_1 : \overline{\theta}_2)^{t_1,\dots,t_k}$. Let $\rho = (\theta_1 : \theta_2)^{t_1,\dots,t_k}$ be the reference corresponding to $\overline{\rho}$ in $\mathscr{F}$, and let $a_1 : a_2$ be the range of t_c in $\mathscr{F}$. (The a_1 is the same because $\overline{\mathscr{F}}$ is a generalisation of $\mathscr{F}$.) Let $r^* = \text{Res}(\theta_1, a_1) : \text{Res}(\theta_2, a_2)$ be the overall target range of ρ. If a is a cell address, let $R(a)$ be the cell in r^* nearest to a (a itself if a is in r^*), and let $t(a)$ be the target tile of ρ that covers $R(a)$. Of course, the target tiles of ρ correspond to those of $\overline{\rho}$.

Lemma 2. *For any cell a, no target tile t' of $\phi(\overline{\rho})$ other than possibly $t(a)$ covers a in $\phi(\overline{\mathscr{F}})$. Thus, the target tiles of $\phi(\overline{\rho})$ do not overlap.*

Proof We know that t' overlaps r^* in $\mathscr{F}$. However, because $t' \neq t(a)$ and tiles do not overlap in $\mathscr{F}$, t' does not cover $R(a)$ in $\mathscr{F}$. That means it must be above, below, or to the left or right of $R(a)$ (or more than one of these).

- If t' is below $R(a)$, then because t' overlaps r^* in at least one row, $R(a)$ must not be in the bottom row of r^*. By the definition of $R(a)$, a must be in the same row as $R(a)$ or above. But the top row of t' in $\overline{\mathscr{F}}$ is the same constant as in $\mathscr{F}$ by regularity, so a must be above t' in $\phi(\overline{\mathscr{F}})$.
- If t' is above $R(a)$, then because t' overlaps r^* in at least one row, $R(a)$ must not be in the top row of r^*. By the definition of $R(a)$, a must be in the same row as $R(a)$ or below. Since t' is above $R(a)$, it cannot overlap the last row of r^*, so t' is vertically non-final, so its last row in $\overline{\mathscr{F}}$ is the same constant as in $\mathscr{F}$. Thus a is below t' in $\phi(\overline{\mathscr{F}})$.
- The horizontal cases are symmetrical.

■

Lemma 3. *If a is in the calling range $\phi(a_1 : \overline{a}_2)$, then* $\mathrm{Res}(\phi(\overline{\rho}), a)$ *has non-negative height and width.*

Proof We prove that the height is non-negative; the argument for the width is symmetrical. Let $\overline{\sigma} = \overline{\chi}_1 : \overline{\chi}_2$ be the row span reference of $\overline{\rho}$, and let $\sigma = \chi_1 : \chi_2$ be the row span reference of ρ. We need to prove that $\mathrm{Res}(\phi(\overline{\sigma}), a)$ has non-negative height. Let t^* be the tile that covers the lower right corner of r^* in $\mathscr{F}$; it is both vertically and horizontally final. Since $\overline{\rho}$ is well-behaved, $\overline{\chi}_1$ and $\overline{\chi}_2$ have a valid pair of relationships to t^*. The proof proceeds by case analysis on this pair. Note that the Inelastic relationship, which requires t^* to have constant height, cannot be paired with Start, Lockstep, or End, which require t^* to have variable height. Also note that $\overline{\chi}_i = \chi_i$ unless $\overline{\chi}_i$ has the End relationship to t^*.

- (Inelastic, Inelastic): In this case, $\overline{\sigma} = \sigma$. If t_c has fixed height in $\overline{\mathscr{F}}$, then a is in the row span of t_c in $\mathscr{F}$, so $\mathrm{Res}(\phi(\overline{\sigma}), a) = \mathrm{Res}(\sigma, a) = \mathrm{Res}(\sigma, a')$, where a' is an arbitrary cell in t_c in $\mathscr{F}$ that is in the same row as a. By our assumption that $\mathscr{F}$ is closed, $\mathrm{Res}(\sigma, a')$ has nonnegative height. (We cannot use a directly because it may be outside the column span of t_c in $\mathscr{F}$ due to horizontal elasticity, but only its row matters for resolving a row span.) If t_c has variable height in $\overline{\mathscr{F}}$, then χ_1 and χ_2 must both be absolute, so $\mathrm{Res}(\phi(\overline{\sigma}), a) = \mathrm{Res}(\sigma, a_1)$, which again has non-negative height by our assumption that $\mathscr{F}$ is closed.
- (Start, Start): Same argument as (Inelastic, Inelastic) when χ_1 and χ_2 are both absolute.
- (Lockstep, Lockstep): We know $\overline{\sigma} = \sigma$, and $\mathrm{Res}(\sigma, a_1)$ has nonnegative height because $\mathscr{F}$ is closed. Suppose a is n rows below a_1 ($n \geq 0$). Then $\mathrm{Res}(\chi_i, a)$ is n rows below $\mathrm{Res}(\chi_i, a_1)$ for each i, so the height of $\mathrm{Res}(\sigma, a)$ is the same as that of $\mathrm{Res}(\sigma, a_1)$, and thus nonnegative.
- (Start, Lockstep): Same as (Lockstep, Lockstep) except that $\mathrm{Res}(\chi_2, a)$ moves down and $\mathrm{Res}(\chi_1, a)$ does not, so the height of $\mathrm{Res}(\sigma, a)$ is at least that of $\mathrm{Res}(\sigma, a_1)$, and thus nonnegative.

- (Lockstep, End): Since t^* was chosen as a vertically final tile of the overall target range r^*, $\mathrm{Res}(\chi_1, a_2)$ is at or above the bottom of t^* in $\mathscr{F}$. In $\overline{\mathscr{F}}$ and all instantiations thereof, the heights of t_c and t^* must differ by the same constant as in $\mathscr{F}$. So if in $\phi(\overline{\mathscr{F}})$, both t_c and t^* change in height by $x \in \mathbb{Z}$ compared to their heights in $\mathscr{F}$, then both $\mathrm{Res}(\chi_1, \phi(\overline{a}_2))$ and the bottom of t^* move x rows down. Thus, $\mathrm{Res}(\chi_1, \phi(\overline{a}_2))$ is still at or above the bottom of t^*. For arbitrary a in t_c, $\mathrm{Res}(\chi_1, a)$ is at or above $\mathrm{Res}(\chi_1, \phi(\overline{a}_2))$, which (as just observed) is at or above the bottom of t^*, which equals $\mathrm{Res}(\phi(\overline{\chi}_2), a)$. Hence $\mathrm{Res}(\phi(\overline{\sigma}), a)$ has nonnegative height.
- (End, End): $\mathrm{Res}(\phi(\overline{\sigma}), a)$ is just the bottom row of t^*, which has height 1.
- (Start, End): Suppose χ_1 points n rows ($n \geq 0$) above the top row of t^*. We know $\mathrm{Res}(\phi(\overline{\chi}_2), a)$ points to the bottom row. t^* has nonnegative height and the height of $\mathrm{Res}(\phi(\overline{\sigma}), a)$ is greater by n.
- (End, Lockstep) and (Lockstep, Start): These pairs are disallowed by the definition of a well-behaved reference.
- (End, Start): This pair cannot occur. If t^* has height h in $\mathscr{F}$, and χ_1 points to the bottom row of t^* while χ_2 points n rows ($n \geq 0$) above the top of t^*, then σ has height $2 - h - n$. Since $h \geq 2$ in order for t^* to be vertically elastic in $\overline{\mathscr{F}}$, σ has nonpositive height at every cell in the calling tile in $\mathscr{F}$, so $\mathscr{F}$ fails to be nondegenerate.

■

Define the overall target range of $\overline{\rho}$ to be $\overline{r}^* = \mathrm{Res}(\overline{\theta}_1, a_1) : \mathrm{Res}(\overline{\theta}_2, \overline{a}_2)$. Clearly, for every a in $\phi(a_1 : \overline{a}_2)$, $\mathrm{Res}(\phi(\overline{\rho}), a)$ is within $\phi(\overline{r}^*)$. So we just need to show that $\phi(\overline{r}^*)$ is covered by the target tiles.

Lemma 4. *Every cell a in $\phi(\overline{r}^*)$ is covered by the tile $t(a)$ in $\phi(\overline{\mathscr{F}})$.*

Proof We prove that a is in the row span of $t(a)$; the argument for the column span is symmetrical. As before, let $\overline{\chi}_1 : \overline{\chi}_2$ be the row span reference of $\overline{\rho}$ and $\chi_1 : \chi_2$ be that of ρ. Let t' be the vertically final tile that covered the column of $R(a)$ in $\mathscr{F}$.

If a is above r^* (so $R(a)$ is in the top row of r^*), that means $\phi(\mathrm{Res}(\overline{\chi}_1, a_1)) < \mathrm{Res}(\chi_1, a_1)$. Given that there is a length substitution ϕ_0 that specialises $\overline{\mathscr{F}}$ back to $\mathscr{F}$, the only way this can happen is if $\overline{\chi}_1$ contains a length variable α and $\phi(\alpha) < \phi_0(\alpha)$. And the only way $\overline{\chi}_1$ can contain a length variable is if its relationship to t' is End. Since $\overline{\chi}_1$ represents the top endpoint of a reference, the definition of well-behavedness requires that t' have positive height for all length assignments. Furthermore, as we have seen in Lemma 3, if $\overline{\chi}_1$ has the End relationship to t', so must $\overline{\chi}_2$, so they both point to the bottom row of t'. Thus, the row of a can only be $\mathrm{Res}(\phi(\overline{\chi}_1), a_1) = \mathrm{Res}(\phi(\overline{\chi}_2), a_1)$, which is the last row of t', which is in the row span of t' because t' has positive height. Even in $\mathscr{F}$, χ_1 and χ_2 must have pointed to the same row, so r^* has only one row, so $t' = t(a)$. We have shown that a is in the row span of $t(a)$, as desired.

Otherwise, a is at or below the top of r^*, so it is at or below $R(a)$. The (constant) top of $t(a)$ must be at or above $R(a)$, so a is at or below the top of $t(a)$ in $\phi(\overline{\mathscr{F}})$; we only need to prove it is at or above the bottom.

If $t' \neq t(a)$, then a is above the bottom of r^*, so it is on the same row as $R(a)$. Furthermore, $t(a)$ is vertically non-final and has the same row span in $\phi(\overline{\mathscr{F}})$ as in $\mathscr{F}$, which contained $R(a)$, hence a is in the row span of $t(a)$, as desired.

Otherwise, $t' = t(a)$. We know a is within the bottom row of $\phi(\overline{r}^*)$, which is $\phi(\text{Res}(\overline{\chi}_2, \overline{a}_2))$, so it suffices to show that $\phi(\text{Res}(\overline{\chi}_2, \overline{a}_2))$ is within the bottom row of $t(a)$. We proceed by case analysis of the relationship of $\overline{\chi}_2$ to $t(a)$, using some of the same reasoning as in Lemma 3:

- Inelastic: Here $\overline{\chi}_2 = \chi_2$. If t_c has fixed height in $\overline{\mathscr{F}}$, then $\overline{a}_2 = a_2$ and $\phi(\text{Res}(\overline{\chi}_2, \overline{a}_2)) = \text{Res}(\chi_2, a_2)$, which is the last row of r^*. If t_c has variable height in $\overline{\mathscr{F}}$, then $\overline{\chi}_2$ is absolute, so $\phi(\text{Res}(\overline{\chi}_2, \overline{a}_2)) = \text{Res}(\chi_2, a_2)$ even though $\overline{a}_2 \neq a_2$. Since $t(a)$ is vertically final, it must cover this row $\text{Res}(\chi_2, a_2)$ in $\mathscr{F}$, and since its height is fixed, it covers this row in $\overline{\mathscr{F}}$ also.
- Start: Here $\overline{\chi}_2$ points at a constant offset $n \geq 0$ above the top row of $t(a)$. The highest the "bottom row" of $t(a)$ can be is one row above its top row (if $t(a)$ has height zero), and a reference with $n > 0$ will be within this bottom row. If $n = 0$, then the definition of well-behavedness requires that $t(a)$ always have positive height, so the bottom row is no higher than the top row and again $\overline{\chi}_2$ is within the bottom row.
- Lockstep: Since $t(a)$ is vertically final, $\text{Res}(\chi_2, a_2)$ must be at or above the bottom row of $t(a)$ in $\mathscr{F}$. In $\overline{\mathscr{F}}$ and all instantiations thereof, the heights of t_c and $t(a)$ must differ by the same constant as in $\mathscr{F}$. So if in $\phi(\overline{\mathscr{F}})$, both t_c and $t(a)$ change in height by $x \in \mathbb{Z}$ compared to their heights in $\mathscr{F}$, then both $\text{Res}(\chi_2, \phi(\overline{a}_2))$ and the bottom of $t(a)$ move x rows down. Thus, $\text{Res}(\chi_2, \phi(\overline{a}_2))$ is still at or above the bottom of $t(a)$.
- End: $\overline{\chi}_2$ points to the bottom row of $t(a)$, which is within $t(a)$.

■

These three lemmas together prove that $\overline{\rho}$ meets the conditions for unambiguity.

■

8.4 Constraint generation for references

For each range reference $\overline{\rho}$ in $\overline{\mathscr{F}}_0$, the new constraint generation procedure is as follows:

1. Constrain the bottom-row delta variable of each vertically non-final target tile of $\overline{\rho}$ equal to 0.
2. Let t_c be the calling tile of $\overline{\rho}$ and let $\overline{\sigma} = \overline{\chi}_1 : \overline{\chi}_2$ be its row span reference. For $i = 1, 2$, let $\mu_i\{m_i + \hat{\alpha}_i\} = \overline{\chi}_i$. For each vertically final target tile t_t of $\overline{\rho}$ and for $i = 1, 2$, let $\hat{\alpha}_t$ and $\hat{\alpha}_c$ be the bottom-row delta variables of t_t and t_c, and try the cases below in order for the relationship of $\overline{\chi}_i$ to t_t. However, if the pair of cases for $i = 1, 2$ for the same target tile t_t would be either (Lockstep, Start) or (End, Lockstep) in that order, then use (Inelastic, Inelastic) instead.

   a. *(Lockstep)* If μ_i is relative and the heights of t_t and t_c in $\mathscr{F}$ are at least 2, then we anticipate that the relationship is Lockstep, but it may turn out to be Inelastic. Constrain $\hat{\alpha}_t = \hat{\alpha}_c$ and $\hat{\alpha}_i = 0$.

b. *(Start)* Otherwise, if all of the following conditions hold:

  * The height of t_t in $\mathscr{F}$ is at least 2.
  * μ_i is absolute or the height of t_c in $\mathscr{F}$ is 1.
  * m_i points at or above the top row of t_t in $\mathscr{F}$.

  Then we anticipate that the relationship is Start, but it may turn out to be Inelastic. Perform the following steps:

  * Constrain $\hat{\alpha}_i = 0$.
  * If m_i points exactly to the top row of t_t in $\mathscr{F}$ and $i = 2$, then constrain the height of t_t to be at least 1.

c. *(End)* Otherwise, if all of the following conditions hold:

  * The height of t_t in $\mathscr{F}$ is at least 2.
  * μ_i is absolute or the height of t_c in $\mathscr{F}$ is 1.
  * m_i points to the bottom row of t_t in $\mathscr{F}$.

  Then we anticipate that the relationship is End, but it may turn out to be Inelastic. Perform the following steps:

  * Constrain $\hat{\alpha}_i = \hat{\alpha}_t$.
  * If $i = 1$, then constrain the height of t_t to be at least 1.

d. *(Inelastic)* Otherwise, the relationship must be Inelastic. Constrain $\hat{\alpha}_t = \hat{\alpha}_i = 0$. In addition, if μ_i is relative, then constrain $\hat{\alpha}_c = 0$.

3. Follow the analogues of steps (1) and (2) for the column axis.

8.5 Correctness of the inference algorithm

Finally, we update the proof of correctness of the inference algorithm (Section 7.4). The first issue is the requirement of the constraint solver that every delta variable is transitively constrained equal to zero or to a delta variable with a lower bound. As before, the main algorithm takes care of this for tile delta variables, and every reference delta variable is constrained equal to either zero or a tile delta variable, satisfying the requirement.

Now, we prove that the reference constraints characterise well-behavedness of references:

Lemma 5. *Let Θ be a delta substitution such that $\Theta(\overline{\mathscr{F}}_0)$ is an elastic SDF $\overline{\mathscr{F}}$ that is a semi-regular generalisation of $\mathscr{F}$. Let $\overline{\rho}_0$ be a reference in $\overline{\mathscr{F}}_0$, and let $\overline{\rho} = \Theta(\overline{\rho}_0)$ be the corresponding reference in $\Theta(\overline{\mathscr{F}}_0)$. Then Θ satisfies the constraints generated for $\overline{\rho}_0$ if and only if $\overline{\rho}$ is well-behaved.*

Proof The structure of the proof is generally similar to that of Lemma 1. Again, we do the proof for rows only. The first difference is that $\overline{\rho}_0$ may have many target tiles; both well-behavedness and constraint generation are independent for each target tile, so we can prove the equivalence for each tile. For a vertically non-final target tile t_t with bottom-row delta variable $\hat{\alpha}_t$, it's obvious that t_t has constant height if and only if the constraint $\hat{\alpha}_t = 0$

is satisfied. So consider a vertically final target tile t_t. Let $\overline{\sigma}_0 = \overline{\chi}_{01} : \overline{\chi}_{02}$ be the row span reference of $\overline{\rho}_0$, and let $\overline{\sigma} = \overline{\chi}_1 : \overline{\chi}_2$ be the row span reference of $\overline{\rho}$.

For now we pretend that neither the definition of well-behavedness nor the constraint generator has the special restriction on the (Lockstep, Start) and (End, Lockstep) pairs (the "forbidden pairs"); we address that restriction at the end.

We can prove sub-lemmas analogous to the previous ones:

1. If $\overline{\chi}_i$ has the Inelastic relationship to t_t, then it satisfies the constraints no matter which case the constraint generator uses for $\overline{\chi}_{0i}$.
2. If $\overline{\chi}_i$ has the Lockstep, Start, or End relationship to t_t, then the constraint generator uses the corresponding case for $\overline{\chi}_{0i}$.

We can then prove, as before, that if the constraint generator uses a given case C for $\overline{\chi}_{0i}$:

3. If the constraints for $\overline{\chi}_{0i}$ are satisfied, then $\overline{\chi}_i$ has relationship either C or Inelastic to t_t.
4. If $\overline{\chi}_i$ has a valid relationship to t_t, then the constraints for $\overline{\chi}_{0i}$ are satisfied.

We will need one extra property:

5. If the Inelastic constraints for $\overline{\chi}_{0i}$ are satisfied (even if the conditions for another constraint generation case would be met), then $\overline{\chi}_i$ has relationship Inelastic to t_t.

Now we consider well-behavedness and constraint generation with the restriction on forbidden pairs.

- First suppose the constraints are satisfied. If the constraint generator hits a forbidden pair and falls back to (Inelastic, Inelastic), then $\overline{\sigma}$ is well-behaved by (5). If the constraint generator's case pair is not forbidden, then by (3), the relationship pair of $\overline{\sigma}$ is not forbidden either, so $\overline{\sigma}$ is well-behaved.
- For the converse, suppose $\overline{\sigma}$ is well-behaved. Notice that of the possible relationships of $\overline{\chi}_i$ to t_t, Inelastic requires that t_t have constant height, while Lockstep, Start, and End all require that t_t have variable height. So either:
  - $\overline{\chi}_1$ and $\overline{\chi}_2$ are both Inelastic. Then the constraints are satisfied by (1) regardless of the cases used by the constraint generator.
  - $\overline{\chi}_1$ and $\overline{\chi}_2$ both have non-Inelastic relationships. By (2), the constraint generator initially attempts to use the corresponding cases. Since $\overline{\sigma}$ is well-behaved, its relationships do not constitute a forbidden pair, so the constraint generator does not fall back and we use (4) as usual to show that the constraints are satisfied.

■

The remainder of the proof of correctness of the inference algorithm (Theorem 2) goes through as before, and we have:

Theorem 4. *With reference well-behavedness defined by Definition 8, every tame SDF $\mathscr{F}$ has a principal regular generalisation $\overline{\mathscr{F}}^*$, and the algorithm in Section 7.1 (using the reference constraint generator of Section 8.4) finds it.*

9 Translation of elastic SDFs to executable form

In Section 4.2 we sketched the semantics of an elastic SDF, but doing so relied on an execution model that uses the target-tile label on each reference to disambiguate references. One could imagine an implementation based directly on this model, but in this section we sketch three alternative routes for execution: using multiple worksheets (Section 9.1), using coordinate arithmetic to avoid tile overlaps (Section 9.2), or using array-level operations instead of element-level ones (Section 9.3). A detailed description and evaluation of these implementation strategies is beyond the scope of this paper. Indeed, spreadsheet tools support various tricky formula constructs (for example, functions such as ROW and COLUMN that get the calling row and column number) that may require care to emulate correctly under each of these strategies. Our purpose here is to reassure the reader that elastic SDFs can be implemented efficiently, and to provide background on the prototype used in our user study.

Given that implementation techniques exist for SDFs (Sestoft, 2014), the key new challenge for elastic SDFs is to avoid overlap between tiles when instantiating the length variables. Indeed, we define:

Definition 9 (Overlap-free elastic SDF)**.** *An elastic SDF $\overline{\mathscr{F}}$ is* overlap-free *if no two of its tiles overlap for any length assignment.*

9.1 Use multiple worksheets to avoid tile overlaps

One way to make an elastic SDF overlap-free is to use multiple worksheets. Indeed, if the SDF is basic (each reference has exactly one target tile), then the translation is easy: we simply place each tile on a separate worksheet. For example, here is a translation of our SHOP SDF, using the standard "!" notation for worksheet references:

```
function SHOP( $S_1$!F4:F{3+$\alpha$}, $S_2$!G2 ) returns $S_5$!H7 {
  $S_3$!G4:G{3+$\alpha$} = $S_1$!F4 * $S_2$!$G$2
  $S_4$!H4:H{3+$\alpha$} = $S_1$!F4 + $S_3$!G4
  $S_5$!H7 = SUM($S_4$!H4:H{3+$\alpha$})  /* No overlap because tiles are
                                         on separate sheets $S_4$, $S_5$ */
}
```

If $\overline{\mathscr{F}}$ is not basic, we can use a slightly more complex process: move each tile to a separate worksheet, then for each range reference $\overline{\rho} = (\overline{\theta}_1 : \overline{\theta}_2)^{t_1,\dots,t_k}$, generate a worksheet that copies $t_1, \dots, t_k$ from their respective worksheets (remember, they never collide because $\overline{\mathscr{F}}$ is well-defined) and update $\overline{\rho}$ to refer to this worksheet.

9.2 Move tiles to avoid overlap

Another approach is to move tiles within the single worksheet so they do not collide. Compared to using multiple worksheets, this approach has the advantage that the transformed SDF bears a greater resemblance to the original in case the user needs to view it for debugging, although other approaches to debugging elastic SDFs may be better yet. Here is the SHOP example again:

```
function SHOP( F4:F{3+α}, G2 ) returns H{4+α} {
  G4:G{3+α} = F4 ∗ $G$2
  H4:H{3+α} = F4 + G4
  H{4+α} = SUM(H4:H{3+α})   /∗ No overlap because H{4+α}
                               moves down as α increases ∗/
}
```

By anchoring the final tile at H{4+α}, rather than H7 as before, it will move downwards as α increases, avoiding the overlap with the tile anchored at H4. This transformation is nontrivial in general, and is described in Appendix A. In general, it produces an *extended elastic SDF* in which the upper-left corner of a tile may involve a linear combination of length variables, which is not allowed by the grammar of Figure 2. The specific algorithm we choose is rather naive but gives adequate results at least on simple examples.

9.3 Transform to an SDF that uses an array to represent each tile

The two previous approaches transform an elastic SDF into another that is constructed so that overlaps cannot occur: these elastic SDFs can be interpreted according to a simplified form of the semantics of Section 4.2 without references needing to track tiles.

Given a spreadsheet interpreter that supports arrays-in-cells (Blackwell et al., 2004), our third approach is to transform an elastic SDF to an (ordinary) SDF, where we replace each tile with an array in its top-left corner. For example, in the SHOP function, the input vector (of whatever size) can land wholesale in F4, the intermediates (of whatever size) can land in G4 and H4, and no collision arises:

```
function SHOP( F4, G2 ) returns H7 {
/∗ F4 is an array, so the operator ∗ lifts over its elements ∗/
  G4 = F4 ∗ $G$2
/∗ F4 and G4 are arrays, so operator + lifts over both ∗/
  H4 = F4 + G4
  H7 = SUM(H4)   /∗ H4 is an array; SUM adds up its elements ∗/
}
```

Not apparent in this definition is the expectation that the first parameter is a vector and the second is a scalar, but spreadsheets are typically dynamically typed.

The implicit lifting of operators like (+) over arrays (a standard feature of spreadsheets) makes the transformed SHOP function seem particularly simple, but this may not be so for non-basic SDFs. Here is an attempt to write COMPOUND using arrays-as-cell-values.

```
function COMPOUND( A3, F3, F1, A4 ) returns F4 {
  C4 = COLUMN( A4, 1 ) − SHIFT_DOWN( A3, COLUMN( A4, 1) )
  D4 = POWER(1+$F$1, C4)
  E4 = SHIFT_DOWN( F3, F4 ) ∗ D4  /∗ BUT array E4 depends on F4 ∗/
  F4 = E4 + COLUMN(A4,2)           /∗ and F4 depends on E4 ∗/
}
```

While a reference that targets a single tile simply becomes a reference to the corresponding array, a reference that targets multiple tiles becomes a specially constructed array. For instance, the reference A3^{t_s,t_x} in the original code becomes the array SHIFT_DOWN(A3, COLUMN(A4, 1)), where the function COLUMN extracts the first column of the two-column array A4, and the function SHIFT_DOWN inserts the element A3 at the front of the resulting array while dropping its last element. Similarly, the reference $\text{F3}^{t_o,t_4}$ becomes the array SHIFT_DOWN(F3, F4).

Unfortunately, we arrive at a cyclic dependency between arrays E4 and F4: each depends on the other. There is a dependence between items in columns E and F in the original example, but since the items are in separate cells, the spreadsheet interpreter can select a raster-scan schedule without any cycles. By placing these columns within arrays, the interpreter is forced to evaluate one or the other column first, leading to a loop.

To solve this problem, the interpreter in our prototype uses a lazy array to represent each tile. Each lazy array consists of an array of thunks, each a memoized argumentless lambda (lambdas being an experimental feature in our prototype). We obtain a uniform translation of elastic SDFs that gave no problems on functions authored by our users.

As an alternative to lazy arrays, we can introduce array-processing functions that capture particular schedules such as raster scan, as illustrated by the following (correct) version of COMPOUND:

```
function COMPOUND( A3, F3, F1, A4 ) returns F4 {
  C4 = COLUMN( A4, 1 ) − SHIFT_DOWN( A3, COLUMN( A4, 1) )
  D4 = POWER(1+$F$1, C4)
  F4 = VSCAN2( D4, COLUMN( A4, 2 ), F3,
               LAMBDA( bal, int, amt, bal ∗ int + amt ) )
}
```

The function VSCAN2 runs down two arrays in parallel, with an accumulator, and we used a lambda-expression too. We conjecture that a large class of elastic SDFs can be efficiently implemented by transformation to concrete SDFs using arrays-as-cell-values, with appeal to implicit lifting and to explicit array-functions like COLUMN and VSCAN2. Moreover, we believe that the manual use of these techniques represents the best solution, using previously known spreadsheet technology, to the problem of writing SDFs that act on variable-sized arrays. The point of our user study is to test elastic SDFs versus this alternative technology with a group of spreadsheet users.

Can we automatically transform an elastic SDF to use array-processing functions? We believe so, as long as it has a simple schedule that we can identify, but leave the details to future work.

10 User Study

We believe that SDFs are the best way to provide user-defined functions to spreadsheet programmers, but any viable solution must work for variable-length inputs. How would users write SDFs for variable-length inputs, if elastic SDFs were unavailable? The most plausible alternative is to write the SDF using arrays-at-a-time operations, rather than element-wise operations. This approach is described, with a user study, by Blackwell et al. (2004). There is already limited support for arrays-at-a-time operations in various spreadsheet packages.[24] The implementation of array programming used in our study was very

[24] https://aka.ms/excel-dynamic-arrays; https://support.google.com/docs/answer/3093275?hl=en; last accessed: January 6, 2020

similar to the dynamic arrays feature of Excel, with the addition of raster scanning functions such as VSCAN, which are not currently supported in Excel's implementation, but which were necessary for completing the real-world representative tasks in our user study.

We therefore ask: from the user's point of view, for a task involving a variable-length input SDF, are elastic SDFs better or worse than using arrays-at-a-time operations? Concretely, we designed a user study to investigate the following:

- RQ1: Does the use of elastic SDFs versus array programming affect the cognitive load experienced by users in writing SDFs?
- RQ2: Does the use of elastic SDFs versus array programming affect the subjective user experience?
- RQ3: Are any observed differences affected by users' programming expertise?

There are a number of other very good questions that might be asked, such as:

- Isolating the inference algorithm: do users prefer to intervene in the generalisation process themselves, perhaps by reviewing the inferred constraints, or are they happy leaving this task to be completely automated? The former might make it easier to ensure that the generalisation was performed as expected.
- Expressiveness: are the constraints expressible in the type system able to capture the constraints that (expert) users would exploit?

While we gain a partial understanding of these additional questions through our user study, we do not address them specifically, choosing to focus on comparing elastic SDFs with arrays-at-a-time operations to make the scope of the study tractable. In particular, the design of an interactive tool that allows users to understand and intervene in the elasticisation process is a substantial undertaking, that we leave to future work.

10.1 Prototype of SDFs for User Study

We adapted an existing research prototype of sheet-defined functions written as an add-in for Excel. The prototype already supports lambda-abstractions, and we extended it to support arrays-in-cells and provided functions for manipulating arrays, as outlined in Section 9.3.

To support elastic SDFs, we added a check-box for the user to indicate that an SDF should be elastic. If so, we generalise the given SDF to an elastic SDF, which is then implemented as a concrete SDF by the lazy array translation described in Section 9.3.

10.2 Participants

We had twenty participants (seven female) aged 18-35 (mean 24), an above-average sample for within-subjects experiments in human-computer interaction (Caine, 2016). They were students in Statistical Science, Mathematics, Management, Computer Science and related disciplines. Ten participants also had industry experience working with spreadsheets. Participants were recruited through convenience sampling via email, and were selected based on their responses to a screening questionnaire which asked them to self-assess their spreadsheet and programming experience. To qualify, participants needed to be familiar with using formulas and ranges in spreadsheets. All participants had some (self-assessed) expertise with spreadsheets, and some (self-assessed) programming expertise. Participants were reimbursed with a £40 voucher after completing the study.

10.3 Tasks

We developed tasks that were representative of real-world spreadsheet tasks, to maintain external validity (i.e., the validity of applying the conclusions of a scientific study outside the context of that study (Mitchell and Jolley, 2012)). We achieved this by adapting real spreadsheets that we had previously gathered from participants of a different study, in which we had interviewed users who had shared and explained the use and structure of these spreadsheets. We adapted the sheets into tasks by removing personally identifiable information and intellectual property, and then designating a part of the sheet to be converted into an SDF and reused elsewhere. For each task, the participant was presented with a spreadsheet partially filled with fictional data, a brief description of what the sheet was to be used for, and a description of what the participant had to calculate.

Participants were given six spreadsheet tasks. Shortened versions of the task descriptions are included below. The solutions of Task 1 and 6 are included to illustrate how SDFs were made using elastic SDFs and array programming.

1. Claiming expenses.

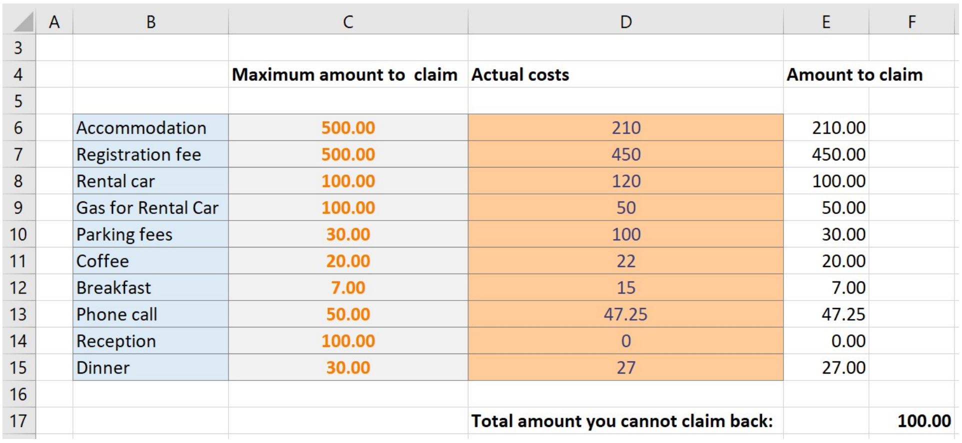
   You have recently come back from a business trip, and now want to claim back the expenses you made. The spreadsheet shows all the expenses incurred for this trip. For each expense, it shows the maximum amount you are allowed to claim back, and the actual costs you incurred. If your actual costs are lower than the maximum amount, you can claim back your actual costs, however if your actual costs exceed the maximum amount, you are only allowed to claim back the maximum amount. Calculate how much money you spent that you cannot claim back. That is, if of all these expenses you can claim back £1,000, but you have spent £1,230 in total, there is £230 which you cannot claim back.

   Figure 11 shows a spreadsheet that computes, in E17, the total amount of money that cannot be claimed back, after calculating for each expense how much can be claimed back in E6:E15. In Calculation View form, the elastic SDF is written as the following.

   function EXPENSES(t_1 C6:C$\{5+\alpha\}$, t_2 D6:D$\{5+\alpha\}$) returns F17^{t_4} {
     t_3 E6:E$\{5+\alpha\}$ = IF(D6^{t_2}<C6^{t_1}, D6^{t_2}, C6^{t_1})
     t_4 F17 = SUM(D6:D$\{5+\alpha\}^{t_2}$) − SUM(E6:E$\{5+\alpha\}^{t_3}$)
   }

   This example needs only the simpler system based on the definition of well-behaved references in Section 6.9.
   The array SDF is similar and written as:

   ```
   function ARRAY_EXPENSES( C6, D6 ) returns F17 {
     E6 = IF(D6<C6, D6, C6)
   ```



	A	B	C	D	E	F
3						
4			**Maximum amount to claim**	**Actual costs**	**Amount to claim**	
5						
6		Accommodation	**500.00**	210	210.00	
7		Registration fee	**500.00**	450	450.00	
8		Rental car	**100.00**	120	100.00	
9		Gas for Rental Car	**100.00**	50	50.00	
10		Parking fees	**30.00**	100	30.00	
11		Coffee	**20.00**	22	20.00	
12		Breakfast	**7.00**	15	7.00	
13		Phone call	**50.00**	47.25	47.25	
14		Reception	**100.00**	0	0.00	
15		Dinner	**30.00**	27	27.00	
16						
17				**Total amount you cannot claim back:**		**100.00**

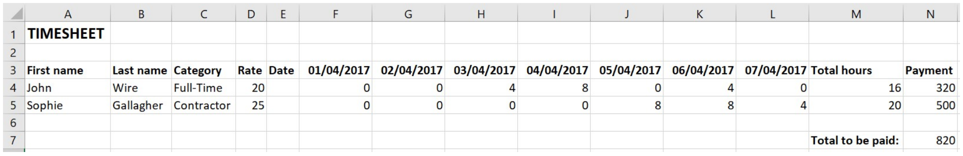
Fig. 11: A spreadsheet containing the definition of the EXPENSES SDF.

F17 = SUM(D6) − SUM(E6)
}

2. Salary payment.

   You are the manager of a school and oversee the hours that all employees have worked, to calculate how much they should be paid. The spreadsheet shows a timesheet, with a row for each employee, and their hourly rate, and their worked hours. Calculate how much salary should be paid in total, to all employees. Hint: To calculate the salary of one employee, sum up his/her hours and multiply this by their specific hourly rate.

   Figure 12 shows a spreadsheet that computes the desired result in N7.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	**TIMESHEET**													
2														
3	**First name**	**Last name**	**Category**	**Rate**	**Date**	**01/04/2017**	**02/04/2017**	**03/04/2017**	**04/04/2017**	**05/04/2017**	**06/04/2017**	**07/04/2017**	**Total hours**	**Payment**
4	John	Wire	Full-Time	20		0	0	4	8	0	4	0	16	320
5	Sophie	Gallagher	Contractor	25		0	0	0	0	8	8	4	20	500
6														
7													**Total to be paid:**	820

Fig. 12: A spreadsheet containing the definition of the SALARYPAYMENT SDF.

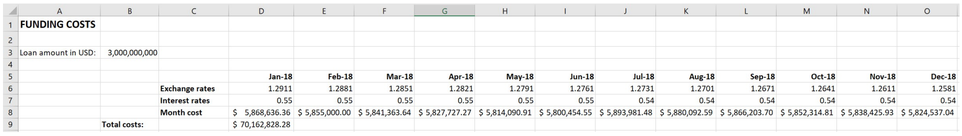
3. Bank loan.

   You are working at a bank, and are considering to extend a loan in US dollars to a client over a fixed time period. The spreadsheet shows the amount you plan to lend in US dollars, the expected exchange rates for each month, and the expected interest rates for each month. Calculate your total expected costs over the time period.

   Figure 13 shows a spreadsheet that computes the desired result in D9.

4. Worked hours.

   You use a spreadsheet to keep track of how many hours you should work per day each month, and how many hours you have actually worked. The spreadsheet shows, for each month: how many hours you have been paid and thus how many hours you should work for the whole month, how many hours you have worked each day so far,

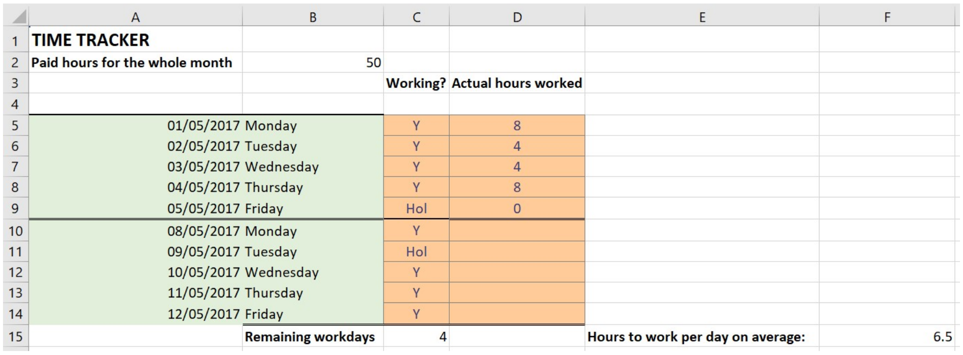


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	**FUNDING COSTS**														
2															
3	Loan amount in USD:	3,000,000,000													
4															
5				**Jan-18**	**Feb-18**	**Mar-18**	**Apr-18**	**May-18**	**Jun-18**	**Jul-18**	**Aug-18**	**Sep-18**	**Oct-18**	**Nov-18**	**Dec-18**
6			**Exchange rates**	1.2911	1.2881	1.2851	1.2821	1.2791	1.2761	1.2731	1.2701	1.2671	1.2641	1.2611	1.2581
7			**Interest rates**	0.55	0.55	0.55	0.55	0.55	0.55	0.54	0.54	0.54	0.54	0.54	0.54
8			**Month cost**	$ 5,868,636.36	$ 5,855,000.00	$ 5,841,363.64	$ 5,827,727.27	$ 5,814,090.91	$ 5,800,454.55	$ 5,893,981.48	$ 5,880,092.59	$ 5,866,203.70	$ 5,852,314.81	$ 5,838,425.93	$ 5,824,537.04
9		**Total costs:**		$ 70,162,828.28											

Fig. 13: A spreadsheet containing the definition of the BANKLOAN SDF.

and how many working days you have left in the month. Use the amount of days left in the month that you have to work to calculate how many hours you should work per day on average for the rest of the month.

Figure 14 shows a spreadsheet that computes the desired result in F15.



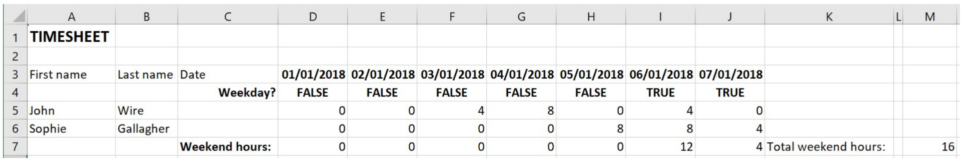
	A	B	C	D	E	F
1	**TIME TRACKER**					
2	**Paid hours for the whole month**	50				
3			**Working?**	**Actual hours worked**		
4						
5	01/05/2017 Monday		Y	8		
6	02/05/2017 Tuesday		Y	4		
7	03/05/2017 Wednesday		Y	4		
8	04/05/2017 Thursday		Y	8		
9	05/05/2017 Friday		Hol	0		
10	08/05/2017 Monday		Y			
11	09/05/2017 Tuesday		Hol			
12	10/05/2017 Wednesday		Y			
13	11/05/2017 Thursday		Y			
14	12/05/2017 Friday		Y			
15		**Remaining workdays**	4		**Hours to work per day on average:**	6.5

Fig. 14: A spreadsheet containing the definition of the WORKINGHOURS SDF.

5. Weekend hours.

You are the manager of a school and oversee the hours that all employees have worked, to calculate how much they should be paid. The spreadsheet shows a timesheet, with a row for each employee and their worked hours. In a previous task, you worked out how much they should be paid. However, a new rule has come into place that employees should be paid more when they have worked on the weekend; therefore, you now want to know how many of the worked hours in the timesheet were on a weekend day. Calculate the total number of worked hours in the weekend.

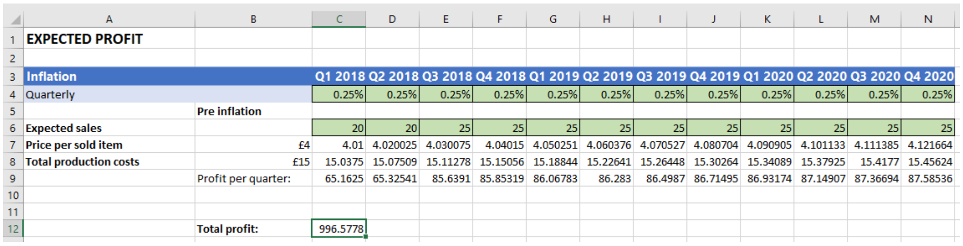
Figure 15 shows a spreadsheet that computes the desired result in M7.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	**TIMESHEET**												
2													
3	First name	Last name	Date	**01/01/2018**	**02/01/2018**	**03/01/2018**	**04/01/2018**	**05/01/2018**	**06/01/2018**	**07/01/2018**			
4			**Weekday?**	**FALSE**	**FALSE**	**FALSE**	**FALSE**	**FALSE**	**TRUE**	**TRUE**			
5	John	Wire		0	0	4	8	0	4	0			
6	Sophie	Gallagher		0	0	0	0	8	8	4			
7			**Weekend hours:**	0	0	0	0	0	12	4	Total weekend hours:		16

Fig. 15: A spreadsheet containing the definition of the WEEKENDHOURS SDF.

6. Expected profit.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	**EXPECTED PROFIT**													
2														
3	**Inflation**		**Q1 2018**	**Q2 2018**	**Q3 2018**	**Q4 2018**	**Q1 2019**	**Q2 2019**	**Q3 2019**	**Q4 2019**	**Q1 2020**	**Q2 2020**	**Q3 2020**	**Q4 2020**
4	Quarterly		0.25%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%
5		**Pre inflation**												
6	**Expected sales**		20	20	25	25	25	25	25	25	25	25	25	25
7	**Price per sold item**	£4	4.01	4.020025	4.030075	4.04015	4.050251	4.060376	4.070527	4.080704	4.090905	4.101133	4.111385	4.121664
8	**Total production costs**	£15	15.0375	15.07509	15.11278	15.15056	15.18844	15.22641	15.26448	15.30264	15.34089	15.37925	15.4177	15.45624
9		Profit per quarter:	65.1625	65.32541	85.6391	85.85319	86.06783	86.283	86.4987	86.71495	86.93174	87.14907	87.36694	87.58536
10														
11														
12		**Total profit:**	996.5778											

Fig. 16: A spreadsheet containing the definition of the PROFIT SDF.

You are the producer of water bottles, and want to estimate your expected profit over several quarters. The spreadsheet shows: a price per bottle, which is expected to change each quarter depending on inflation rate; expected sales, which is the number of bottles you are expected to sell each quarter; expected inflation rate per quarter;, and a fixed production cost that is the same regardless of how many water bottles you sell, but is expected to change each quarter depending on inflation rate. Calculate your total expected profit.

Figure 16 shows a spreadsheet that computes, in C12, the total expected profit. The solution for the elastic SDF and array SDF is given in Calculation View form below. Note: HSCAN is the horizontal analog of VSCAN and was provided to the participants:

```
function HSCAN (A2,B1:{A+$\alpha$}1,f) returns B2:{A+$\alpha$}2 {
  B2:{A+$\alpha$} = f(A2,B1)
}
```

The elastic SDF is written as:

```
function PROFIT( $t_1$ C4:{B+$\alpha$}4, $t_2$ C6:{B+$\alpha$}6 , $t_3$ B7, $t_4$ B8 ) returns C12$^{t_8}$ {
  $t_5$ C7:{B+$\alpha$}7 = B7$^{t_3,t_5}$ $*$ (1 + C4$^{t_1}$)
  $t_6$ C8:{B+$\alpha$}8 = B8$^{t_4,t_6}$ $*$ (1 + C4$^{t_1}$)
  $t_7$ C9:{B+$\alpha$}9 = ( C6$^{t_2}$ $*$ C7$^{t_5}$ ) $-$ C8$^{t_6}$
  $t_8$ C12 = SUM(C9:{B+$\alpha$}9$^{t_7}$)
}
```

This example needs the extended system of Section 8.
The array SDF is written as:

```
function ARRAY_PROFIT( C4,C6, B7, B8 ) returns C12 {
  C7 = HSCAN(LAMBDA(X,Y,X$*$1+Y), B7, C4)
  C8 = HSCAN(LAMBDA(X,Y,X$*$1+Y), B8, C4)
  C9 = (C6 $*$ C7) $-$ C8
  C12 = SUM(C9)
}
```

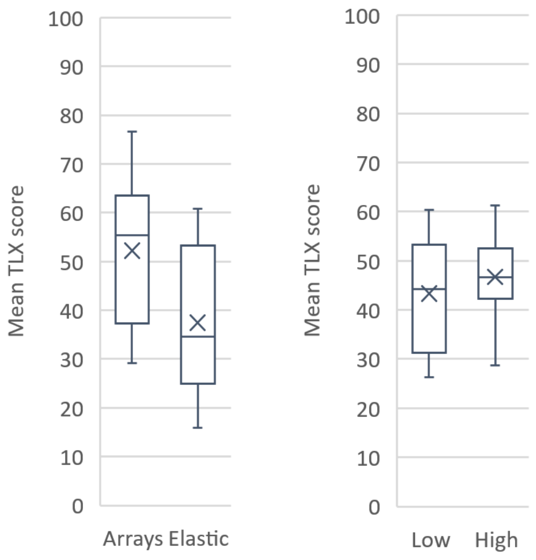
10.4 Protocol

Prior to the study, participants watched a 10-minute instructional video that explained how to create SDFs, how to use elastic SDFs, and our array notation, with step-by-step examples. A study session consisted of two parts, one in which the participant used elastic SDFs, and one in which they used arrays to define SDFs. In each part, a practice task was performed followed by three task trials. The order of conditions was counterbalanced to avoid order effects: one group of participants used arrays in the first part, and the second group of participants used elastic SDFs in the first part. After each part, participants completed a questionnaire to measure their perceived workload. We used the NASA Task Load Index (TLX) questionnaire, a commonly used tool in user-centred design research, which enables users to self-assess their workload during a task (Hart and Staveland, 1988). The questionnaire consists of six subscales: mental demand, physical demand, temporal demand, performance, effort and frustration. The user is asked to rate their subjective workload on each scale, which ranges from 5 (low workload) to 100 (high workload) in increments of 5 (thus effectively a 20-point scale). These ratings can be averaged to yield the overall cognitive load. After the second part, participants were interviewed on their overall experience. The study lasted approximately two hours on average.

10.5 Results

10.5.1 Workload

A mixed analysis of variance (ANOVA) was used to analyse the TLX scores. The ANOVA is a statistical model used to analyse any differences in mean score among groups. A significance level (α) of 0.05 was used; we interpreted p-values lower than this to mean that the observed difference was unlikely due to chance.



(a) Arrays vs elastic SDFs (b) Programming expertise

Fig. 17: Boxplots showing differences in cognitive load scores between groups. (a) Cognitive load was lower with elastic SDFs than with arrays. (b) Cognitive load was not significantly affected by programming expertise.

Participants perceived a significantly lower workload for elastic SDFs (Figure 17a) (M = 37.46, SD = 14.24) than arrays (M = 52.25, SD = 14.80), F(1, 18)[25] = 10.22, $p < 0.01$. There was no significant difference between the group starting with arrays (M = 47.92, SD = 15.08) and the group starting with elastic SDFs (M = 41.79, SD = 17.02), F(1,18) = 1.93, p = 0.2. There was also no significant interaction effect between SDF version and order, F(1,18) = 1.31, p = 0.3. This lack of interaction means that both groups associated elastic SDFs with a lower workload.

Our prototype was built as an Excel add-in, but issues in our implementation (unrelated to the elasticity inference algorithm) created occasional delays in response to user interactions, and instances where Excel needed to be restarted. Due to this limitation, we were unable to draw a statistical comparison of task completion times between elastic SDFs and array programming. Average task completion times for tasks unaffected by technical interruptions are described here to give some insight on the timing advantage of elastic SDFs, but this comparison is not statistically formal. When participants used arrays, the average task completion time was 13 minutes and 34 seconds (814 seconds). Using elastic SDFs, the average task completion time was 7 minutes and 48 seconds (468 seconds).

Participants' solutions were checked for correctness, and tasks were considered completed if they were correct. Of 160 task instances (2 conditions $\times$ 4 tasks $\times$ 20 participants), 33 (21%) were not completed. In only 3 cases, these were not completed because the participant could not solve the problem. All 3 unsolved cases occurred with the array programming condition, but with 3 different participants and 3 different tasks, thus the inability to solve the task cannot be attributed to any individual or the hardness of any particular task. The remaining 30 non-completed tasks were not completed either because of technical difficulties with the prototype (11 cases) or because too much time was spent on earlier tasks and the allotted time for the experiment ran out before the participant could reach later tasks (19 cases). However, every participant successfully completed at least one task with arrays and elastic SDFs respectively, which is sufficient for the validity of the TLX comparison.

10.5.2 Programming experience

We asked participants to rate their programming experience on a scale from 1 to 4, with 1 being 'little to no experience', 2 'some experience, still a beginner', 3 'extensive experience, some expertise' and 4 being 'very experienced, high expertise'. They also stated how many years of programming experience they had. We divided our participants into 'low' and 'high' expertise groups. Low expertise participants classified themselves as a beginner (self-rated experience level of 1 or 2) or had less than two years of experience. Seven participants fell into the low expertise group, 13 in the high expertise group. We did not observe significant differences in cognitive load between low (M = 43.33, SD = 10.51) and high (M = 46.71, SD = 8.61) expertise programmers, F(1,18) = 2.20, p = 0.16 (see Figure 17b). There was also no interaction effect between SDF version or programming

[25] The numbers in brackets indicate the degrees of freedom and are calculated from the number of groups and number of participants of the study. These are used to assess how large the F value needs to be in order to reject the hypothesis that mean scores of different groups are equal. For a detailed explanation of ANOVA results and notation, see Field (2013).

experience, F(1,18) = 0.09, p = 0.77, thus we do not have evidence that the observed differences in cognitive load between elastic SDFs and array programming were affected by users' programming experience. This means that we did not obtain evidence that elastic SDFs offer differing cognitive benefits to novice /low-expertise as well as high-expertise programmers.

10.5.3 Interviews

In the interview, participants were positive about SDFs and could see them being applied to their own work in which custom calculations often have to be re-used: *'Just yesterday I was doing things to calculate the pre-taxed value, after we know the post-taxed value. Then I always just copy and paste the functions, over and over again, because the original Excel file that I'm given is kind of messy. So if I have one function to do that, that would be much easier.'* (P17, accountant).

Arrays were found useful in reducing the manual effort required to repeat a simple, built-in function for a large range of cells. For example, when participants had to do a multiplication for each row in the sheet, instead of having to enter a formula for the first row, and then drag-fill the formula down the range of rows, they only needed to enter the function once if these rows were held in one cell as an array, and the function was automatically lifted over elements of the input array to produce an output array. However, when authoring and invoking complex SDFs, participants preferred ranges to arrays, because they were familiar with passing ranges as arguments into built-in functions. Similarly, participants commented that they preferred elastic SDFs over arrays because the use of ranges was more similar to their typical use of formulas: *'I think elastic functions are easier to work with, also with the mental model that you have of Excel, because you can more just follow your normal Excel workflow.'* (P9, student in Computer Science).

Participants liked that the implementation details of SDFs are hidden when the SDF is invoked, but they also wanted to have the option to see further details (i.e., a trace) at invocation sites. Participants wanted to understand how the function behaved with different types of input, debugging, and to inspect intermediate results: *'This saves a lot of time, but it's always important to check the substeps. There is some trade-off between making steps invisible and not being able to spot mistakes.'* (P13, former economist at a bank).

Furthermore, some participants desired control over which arguments to make elastic, as not all elasticisable arguments might necessarily make sense to elasticise at the domain level, for example when dealing with a contract or time period with a fixed length.

To address our research questions directly:

- RQ1: Elastic SDFs offered reduced cognitive load, and potentially lower authoring time, compared to array programming.
- RQ2: Participants found arrays useful and transparent when mapping a simple formula over a range, but preferred elastic SDFs for calculations involving complex array combinators, because it allowed the use of familiar formula structure and familiar behaviour of built-in functions that take ranges as arguments.
- RQ3: The observed differences were not affected by users' programming expertise, suggesting similar benefits to low-expertise as well as high-expertise programmers.

In summary: the study found that elastic sheet-defined functions can successfully enable end users to define functions that accept variable-length input, without having to write array combinators. We also observed qualitatively that sheet-defined functions can be a valuable tool in spreadsheet users' work.

11 Related work

Generalising an SDF is an example of *program synthesis*, where the task is to synthesise a program from some specification of what the program should do; see Gulwani et al. (2017) for a recent survey. The specification is often partial; a popular choice is to allow the user to supply a set of input/output examples. The field is a very active one and, like our work, is mostly focused on the needs of non-expert end users rather than professional programmers. However, our work seems unusual: rather than use input/output examples, or program skeletons, we directly generalise a single concrete program to one that handles a broader variety of inputs, and we offer provable guarantees that the result is not just *any* generalisation but the most general one possible.

Sheet-defined functions originated in Forms (Ambler, 1987) and have been implemented in various other research systems since then. Sestoft (2014) describes an implementation of sheet-defined functions with first-class arrays (e.g., an array can be the value of a cell) and compilation to the .NET intermediate language. He does not consider synthesis of SDFs by example.

Various spreadsheet tools let a user define a computation on input of one size and have a mechanism to modify the computation to take input of a different size, but the mechanism has to be invoked manually by the user, and while a user can of course copy a computation, there is no means of sharing logic so that computations on different input sizes can be updated together. The most rudimentary mechanism is drag-filling of formulas, which has to be performed once for each group of contiguous aligned tiles in the computation. (Indeed, drag-filling is the typical means by which a user would build an SDF for a certain fixed input size before making the SDF elastic.) Excel also has support for "tables" with a homogeneous formula in each column; adding rows to a table does the equivalent of a drag-fill of the column formulas, and a syntax is provided to reference an entire column of a table for aggregation.

Abraham and Erwig describe spreadsheet "templates" in the ViTSL language (Erwig et al., 2006). Like an elastic SDF, a ViTSL template describes patterns of repeating formulas and can be specialised to any desired number of repetitions, but there are no means of reusing the same template multiple times within a single spreadsheet. Also, the feature sets differ. A ViTSL template can have a group of rows or columns that repeats ("ABABAB") but cannot specify that two separate groups have the same number of repetitions ("AAA BBB"), while the reverse is true of an elastic SDF. In many cases, a sheet designed in one of those ways can be converted to an equivalent sheet designed the other way, though the attractiveness of the designs to a user may differ. As far as we can tell from the formalisation, ViTSL does not support offset references, which is a major limitation compared to elastic SDFs. Similarly, Paine's 'Model Master' language (Paine, 2008) offers a textual notation for describing spreadsheet computations, but its presentation in terms of array formulas and separation from the grid (requiring the use of an auxiliary 'layout' file) make it more suitable for use by expert programmers, rather than non-expert end users. It does not support SDFs.

Tabula (Mendes and Saraiva, 2017) and Object Spreadsheets (McCutchen et al., 2016) are structured spreadsheet tools that let a user construct a computation that applies to variable-size input by writing element-wise formulas and then reuse the computation on several inputs of different sizes by introducing an outer level of structural repetition around it. However, they do not support extracting such a computation from an existing unstructured spreadsheet, nor (currently) packaging such a computation as a function that can be called from anywhere. Furthermore, offset references are awkward to express in these tools.

Our formal semantics appears to be the first denotational semantics for sheet-defined functions. Sestoft (2014) presents a big-step semantics for simplified spreadsheet formulas, but it does not cover the details of dereferencing ranges nor of sheet-defined functions. More recently, Bock et al. (2020) generalize Sestoft's operational semantics to a spreadsheet language with (concrete) sheet-defined functions and arrays, based on the FunCalc system of Sestoft's book. Neither these operational semantics nor our semantics attempts to capture the details of incremental recalc of spreadsheets. A recent work (Mokhov et al., 2018) explores the connection between incremental spreadsheet recalc and build systems.

Although the work of Peyton Jones et al. (2003) was informed by HCI theories of usability (Green and Petre, 1996; Blackwell, 2002), our work appears to be the first report of a study of sheet-defined functions with actual users.

12 Conclusion

No prior work on sheet-defined functions has considered how to author functions that take variable-sized inputs. And yet a great many spreadsheets act on lists of items to be tallied and carried in a multitude of ways. Our work paves the way for end-user programmers to capture such computations as reusable functions, exemplified on short lists, and reliably generalized to work on arbitrary-sized inputs. The time is ripe to introduce elastic SDFs to the world of spreadsheets!

Acknowledgements. We thank Joe McDaid, Claudio Russo, and Neil Toronto for developing the Excel SDF add-in that we adapted for this study. We thank Nada Amin, Tom Ellis, Martin Erwig, Andrey Mokhov, Monal Narasimhamurthy, Lionel Parreaux, Sruti Srinivasa Ragavan, Peter Sestoft, Dimitrios Vytiniotis, and Jack Williams, and anonymous referees for their feedback on drafts of this paper.

Conflicts of interest. During the course of this research, all the authors worked at Microsoft, maker of the Excel spreadsheet product.

References

Ambler, A. 1987. Forms: Expanding the visualness of sheet languages. In *1987 Workshop on Visual Languages*, pp. 105–117. Tryck-Center Linköping.

Blackwell, A. F. 2002. First steps in programming: A rationale for attention investment models. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pp. 2–10. IEEE.

Blackwell, A. F., **Burnett, M. M.**, **and Jones, S. P.** 2004. Champagne prototyping: A research technique for early evaluation of complex end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pp. 47–54. IEEE.

Bock, A. A., **Bøgholm, T.**, **Sestoft, P.**, **Thomsen, B.**, **and Thomsen, L. L.** 2020. On the semantics for spreadsheets with sheet-defined functions. *Journal of Computer Languages*, 57.

Bostrom, R. P., **Olfman, L.**, **and Sein, M. K.** 1990. The Importance of Learning Style in End-User Training. *MIS Quarterly*, 14(1):101.

Caine, K. 2016. Local standards for sample size at CHI. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 981–992. ACM.

Erwig, M., **Abraham, R.**, **Kollmansberger, S.**, **and Cooperstein, I.** 2006. Gencel: a program generator for correct spreadsheets. *Journal of Functional Programming*, 16(3):293–325.

Field, A. 2013. *Discovering Statistics Using IBM SPSS Statistics*. Sage Publications Ltd., 4th edition.

Green, T. R. G. and Petre, M. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174.

Gulwani, S., **Polozov, O.**, **and Singh, R.** 2017. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119.

Hart, S. G. and Staveland, L. E. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. *Human mental workload*, 1(3):139–183.

Jansen, B. 2015. Enron versus euses: A comparison of two spreadsheet corpora. *arXiv preprint arXiv:1503.04055*.

McCutchen, M., **Itzhaky, S.**, **and Jackson, D.** 2016. Object Spreadsheets: A new computational model for end-user development of data-centric web applications. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pp. 112–127, New York, NY, USA. ACM.

Mendes, J. and Saraiva, J. 2017. Tabula: A language to model spreadsheet tables. *CoRR*, abs/1707.02833.

Mitchell, M. L. and Jolley, J. M. 2012. *Research design explained*. Cengage Learning.

Mokhov, A., **Mitchell, N.**, **and Peyton Jones, S.** 2018. Build systems à la carte. *PACMPL*, 2(ICFP):79:1–79:29.

Paine, J. 2008. Ensuring spreadsheet integrity with model master. *arXiv preprint arXiv:0801.3690*.

Pandita, R., **Parnin, C.**, **Hermans, F.**, **and Murphy-Hill, E. R.** 2018. No half-measures: A study of manual and tool-assisted end-user programming tasks in Excel. In *VL/HCC*, pp. 95–103. IEEE Computer Society.

Peyton Jones, S., **Blackwell, A.**, **and Burnett, M.** 2003. A user-centred approach to functions in Excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pp. 165–176, New York, NY, USA. ACM.

Sarkar, A., **Gordon, A. D.**, **Peyton Jones, S.**, **and Toronto, N.** 2018. Calculation view: multiple-representation editing in spreadsheets. In *VL/HCC*, pp. 85–93. IEEE Computer Society.

Sestoft, P. 2014. *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press.

Sestoft, P. and Sørensen, J. Z. 2013. Sheet-defined functions: implementation and initial evaluation. In *International Symposium on End User Development*, pp. 88–103. Springer.

A Moving tiles to avoid overlaps: details

In this Appendix, we describe the translation of an elastic SDF $\overline{\mathscr{F}}$ to a overlap-free elastic SDF by moving tiles. The basic idea is simple (we position the tiles dynamically in terms of the length variables to keep them from colliding when the variables increase), but we have to deal with several special cases. The stages of the algorithm (explained in detail in the rest of this section) are:

1. Identify *complexes* of tiles that must remain together.
2. Ensure that the tiles within each individual complex do not collide when length variables increase.
3. Move the complexes apart so they do not collide, and update range references accordingly.

A.1 Identifying complexes

Two tiles in $\overline{\mathscr{F}}$ are *linked* if they both occur in the target-tile set of some range reference $\overline{\rho}$, meaning that we cannot move the tiles to different locations unless we were to replace $\overline{\rho}$ with a more complicated formula to read from both tiles, which the present algorithm does not do. For example, in the COMPOUND elastic SDF of Section 8, tiles t_o and t_4 are linked by the reference F3^{t_o,t_4} in the tile E4:E$\{3+\alpha\}$. A *complex* is a set of tiles that are transitively linked to one another. All target tiles of a given range reference belong to the same complex, so if we move a whole complex by a certain number of rows and columns, we just need to offset all range references to the complex by that number of rows and columns. The first step of the translation is to identify complexes by entering linked tiles in a union-find data structure.

A.2 Ensuring complexes are expandable

A complex is *expandable* if its tiles do not collide for any values of the length variables. (It suffices to imagine setting all length variables to ∞.) Given that we do not separate the tiles of a complex, we need every complex to be expandable. For typical elastic SDFs, all complexes will be expandable, but in pathological cases, we may get a non-expandable complex. In such a case, we remove all length variables involved in the tile overlap from $\overline{\mathscr{F}}$ by setting them to their initial values. Specifically, suppose that elastic tiles t_1 and t_2 with ranges $a_1 : \overline{N}_1\overline{m}_1$ and $a_2 : \overline{N}_2\overline{m}_2$ collide if all length variables are set to ∞. Two rectangles overlap if and only if their row spans overlap and their column spans overlap. We know that t_1 and t_2 do not overlap in the original SDF $\mathscr{F}$, so their row spans must be disjoint or their column spans must be disjoint (or both). If t_1 and t_2 are disjoint in rows but overlap in columns in $\mathscr{F}$, then we remove all length variables that appear in $\overline{m}_1$ and $\overline{m}_2$, which will leave the tiles t_1 and t_2 in $\overline{\mathscr{F}}$ with their initial row spans, which do not overlap. (This change could also affect the column spans if the same variable appeared in both a height and a width, although as mentioned in Section 6.7, this never occurs in a principal regular generalisation in our current system.) Conversely, if t_1 and t_2 are disjoint in columns but overlap in rows in $\mathscr{F}$, then we remove all length variables that appear in $\overline{N}_1$ and $\overline{N}_2$. If t_1 and t_2 are disjoint in *both* rows and columns in $\mathscr{F}$ (e.g., elastic ranges B4:$\{$A+$\alpha\}$4 and D2:D$\{1{+}\beta\}$ with initial values $\alpha = \beta = 2$), then either change would be sufficient to resolve the overlap, but for symmetry, we perform both of them. In this case, after ensuring complex expandability, we cannot claim to have the "principal regular generalisation with expandable complexes", but it's a pathological case anyway.

A.3 Positioning the complexes

Finally, we are ready to position the complexes dynamically so they do not collide. In this section, we assume that elastic coordinates $\bar{x}$ may contain positive linear combinations of length variables, i.e., they are of the form $\{x + \sum_i u_i\alpha_i\}$ where $u_i > 0$.

If $\overline{m}$ and $\overline{m}'$ are elastic axis positions, we say that $\overline{m}$ is *potentially greater than* $\overline{m}'$ if either the constant term or some length variable coefficient of $\overline{m} - \overline{m}'$ is greater than zero. We say that elastic spans $\overline{m}_1 : \overline{m}_2$ and $\overline{m}'_1 : \overline{m}'_2$ *potentially overlap* if $\overline{m}_2$ is potentially greater than $\overline{m}'_1 - 1$ and $\overline{m}'_2$ is potentially greater than $\overline{m}_1 - 1$. Finally, two elastic rectangles *potentially overlap* if their row spans potentially overlap and their column spans potentially overlap. (There are various cases in which two elastic rectangles that potentially overlap do not actually overlap for any length assignment, but that's OK.) The *upper bound* of a set of elastic coordinates is an elastic coordinate that has the maximum of their constant terms and the maximum of their coefficients for each length variable, and the *lower bound* is defined analogously.

The positioning algorithm is as follows: Compute an elastic bounding rectangle for each complex by taking upper and lower bounds of the coordinates of the tiles of the complex. Then do the following for each complex $\mathscr{C}$, where complexes are ordered according to the first occupied cell of each complex and cell addresses are compared lexicographically:

1. Let $\overline{N}\overline{m}$ be the current upper left corner of $\mathscr{C}$'s bounding rectangle, and let S be the set of previously processed complexes whose bounding rectangles potentially overlap that of $\mathscr{C}$. If S is empty, then positioning of $\mathscr{C}$ is complete, otherwise continue.
2. Let S_R be the set of complexes in S whose row spans in $\mathscr{F}$ are disjoint from the row span of $\mathscr{C}$ in $\mathscr{F}$, and define S_C analogously for columns. Let $S' = S \setminus (S_R \cup S_C)$; a complex may belong to S' if its bounding rectangle in $\mathscr{F}$ overlaps that of $\mathscr{C}$ even though we know that the actual tiles do not overlap.
3. Let $\overline{m}'$ be the upper bound of $\overline{m}$ and one more than the bottom bounding edge of each complex in $S_R \cup S'$, and let $\overline{N}'$ be the upper bound of $\overline{N}$ and one more than the right bounding edge of each complex in $S_C \cup S'$.
4. Move $\mathscr{C}$ so that its upper left corner is at $\overline{N}'\overline{m}'$. It is now disjoint from $S_R \cup S'$ in rows and disjoint from $S_C \cup S'$ in columns, so it is disjoint from all complexes in S, but it may collide with other complexes. Return to step 1. (The process must terminate because after $\mathscr{C}$ potentially overlaps a given previously processed complex and is moved either down or to the right, $\mathscr{C}$ is only moved further down or to the right, so it can never potentially overlap the same complex again.)

Once all complexes have been positioned, we update all range references to reflect the new locations of their respective complexes, and we have the overlap-free extended elastic SDF.

Proposition 4. *For every well-defined elastic SDF $\overline{\mathscr{F}}$, tile movement produces a well-defined overlap-free extended elastic SDF $\overline{\mathscr{F}}_{of}$ that is semantically equivalent to some well-defined elastic SDF $\overline{\mathscr{F}}'$ of which $\overline{\mathscr{F}}$ is a generalisation.*

($\overline{\mathscr{F}}'$ is the elastic SDF that we have after making complexes expandable.)