Elastic Sheet-Defined Functions: Generalising Spreadsheet Functions to Variable-Size Input Arrays*

MATT MCCUTCHEN†, Massachusetts Institute of Technology, US
JUDITH BORGHOUTS†, University College London, UK
ANDREW D. GORDON, Microsoft Research, UK and University of Edinburgh, UK
SIMON PEYTON JONES, Microsoft Research, UK
ADVAIT SARKAR, Microsoft Research, UK

Sheet-defined functions (SDFs) bring modularity and abstraction to the world of spreadsheets. Alas, users naturally write SDFs that work over *fixed-size* arrays, which limits their re-usability. We describe a principled approach to *generalising* such functions to become *elastic SDFs* that work over inputs of arbitrary size, including a principal-generalisation theorem and empirical evaluation through a user study.

1 INTRODUCTION

Suppose we want to use a spreadsheet to compute the average of the numeric cells[1] 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 in Sestoft's book [Sestoft 2014]. 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:

```
function AVG⟨α⟩( 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 the 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 it 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:

*Draft as of March 6, 2019; see https://aka.ms/calcintel for the latest version.

†This work was done while these authors were working at Microsoft Research.

[1]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 does this.

Authors' addresses: Matt McCutchen, Massachusetts Institute of Technology, 77 Massachusetts Ave, Cambridge, 02139, US, matt@mattmccutchen.net; Judith Borghouts, University College London, Gower Street, London, WC1E 6BT, UK, judith.borghouts.14@ucl.ac.uk; Andrew D. Gordon, Microsoft Research, 21 Station Road, Cambridge, CB1 2FB, UK, University of Edinburgh, Old College, South Bridge, Edinburgh, EH8 9YL, UK, adg@microsoft.com; Simon Peyton Jones, Microsoft Research, 21 Station Road, Cambridge, CB1 2FB, UK, simonpj@microsoft.com; Advait Sarkar, Microsoft Research, 21 Station Road, Cambridge, CB1 2FB, UK, advait@microsoft.com.

- A recent paper [Sarkar et al. 2018] proposes a simple textual notation for spreadsheet programs (Section 3). We develop this idea further, by introducing new notation for *corner-size ranges*, and 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 so-called *tiles*. We also discuss practical execution mechanisms in Appendix A.
- Our goal is to find a 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. Better still, we give an algorithm that finds it, and prove the algorithm correct. The algorithm is parameterised over the *generalisation system* (Section 7.4) which allows us to readily explore a variety of tradeoffs between expressiveness and complexity.
- The ultimate goal of programming language research is to make human beings more productive. So in Section 8 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 (so-called 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 formulae 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 next.

The supplemental text for this submission includes a series of appendixes (such as Appendix A mentioned above) with additional content and detail.

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 typical spreadsheet products today. Excel allow, so-called *custom functions* to be defined, but you must do so in Visual Basic or Javascript. Similarly, Google Sheets allows custom functions, provided they are 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?

There is clearly very strong demand for custom functions in spreadsheet:

- Excel has long allowed users to write custom functions in VBA (Visual Basic for Applications; an implementation of the programming language Visual Basic 6). This feature is immensely popular, with the query "Excel VBA" currently retrieving over 100,000 results on StackOverflow.[2] For context: at present, that is greater than the number of results for 7 out of the 20 most popular programming languages (as ranked by the November 2018 Tiobe index[3]).
- Besides writing VBA scripts, users commonly extend the function library available in Excel by installing add-ins. For example, one add-in[4] that introduces a number of functions to Excel is extremely popular, with nearly 1 million 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 VBA 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 choices for SDFs

Sestoft took the idea of sheet-defined functions and produced an open-source implementation, along with a book that describes the details [Sestoft 2014; Sestoft and Sørensen 2013]. There are many design choices, including

- *User interface.* Through what user interface does the user define a new function, give it a name, and specify its parameters and results?
- *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 on 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?

[2] https://stackoverflow.com/questions/tagged/excel+vba. Last accessed: March 6, 2019
[3] https://www.tiobe.com/tiobe-index/. Last accessed: March 6, 2019
[4] https://www.asap-utilities.com/asap-utilities-excel-tools-tip.php?tip=259&utilities=97&lang=en_us. Last accessed: March 6, 2019

Length variable	α, β
Column name	$N ::= \mathrm{A} \mid \ldots \mid \mathrm{Z} \mid \mathrm{AA} \mid \mathrm{AB} \mid \ldots$
Elastic column name	$\overline{N} ::= N \mid \{N + \alpha\}$
Elastic axis position	$\overline{m} ::= m \mid \{m + \alpha\} \quad (m \in \mathbb{Z}^{+})$
Address (column, row)	$a ::= \overline{N}\overline{m}$
Span size	$\bar{l} ::= l \mid \{l + \alpha\} \quad (l \in \mathbb{Z}^{\geq 0})$
Range	$r ::= a \mid a_1 : a_2 \mid a :: \{\bar{l}_1, \bar{l}_2\}$
Absolute/relative marker	$\mu, \nu ::= \$ \mid ‘’$
Cell reference	$\theta ::= \nu\overline{N}\mu\overline{m}$
Range reference	$\rho ::= \theta \mid \theta_1 : \theta_2 \mid \theta :: \{\bar{l}_1, \bar{l}_2\}$
Annotated range reference	$\widetilde{\rho} ::= \rho^{t_1, \ldots, t_n}$
Constant	c (either string or number)
Formula	$F ::= \widetilde{\rho} \mid c \mid f(F_1, \ldots, F_n) \quad (f$ either builtin function or an SDF)
Tile name	t
Range assignment	$\mathcal{A} ::= t\ r = F$
Sheet fragment	$\mathcal{S} ::= \mathcal{A}_1, \ldots, \mathcal{A}_n$
SDF definition	$\mathcal{F} ::= \textsf{function}\ f(r_1, \ldots, r_k)\ \textsf{returns}\ r\ \{\ \mathcal{S}\ \}$ (no t annotations)
Labelled SDF definition	$\hat{\mathcal{F}} ::= \textsf{function}\ f(t_1\ r_1, \ldots, t_k\ r_k)\ \textsf{returns}\ r^{t'_1, \ldots, t'_j}\ \{\ \mathcal{S}\ \}$
Elastic SDF definition	$\widetilde{\mathcal{F}} ::= \textsf{function}\ f\langle\alpha_1, \ldots, \alpha_n\rangle(t_1\ r_1, \ldots, t_k\ r_k)\ \textsf{returns}\ r^{t'_1, \ldots, t'_j}\ \{\mathcal{S}\}$

Fig. 1. Spreadsheet syntax, based on A1-style grid references. In the grammars for an SDF definition or a labelled SDF definition, we require that no length variables occur in r, any r_i, or $\mathcal{S}$.

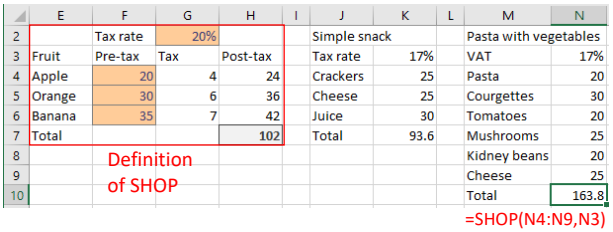
2.3 Elasticity

These are important questions. 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 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.



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

3 A TEXTUAL NOTATION FOR TILES AND SDFS

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. *We do not expect users to use textual notation to define SDFs*; instead, we expect there will be direct UI support for doing so. We believe that there is considerable merit in augmenting the data-centric view offered by typical spreadsheets, by providing a complementary textual calculation view, and 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 a 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.

The complete Calculation View language as used in this paper is given in Figure 1 for reference.

3.1 Range assignments, formulas, and values

Figure 2 shows a spreadsheet that computes, in H7, the total cost of purchasing the items in F4:F6, after accounting for VAT, 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::{3,1})
```

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* (usually omitted in examples), r is a *range*, and F is a *formula*.

Ranges r are already standard in spreadsheets. (For a comprehensive introduction to spreadsheet notation and semantics, see Sestoft's authoritative book [Sestoft 2014].) Referring to Figure 1, a range r can be denoted by:

- A single *cell address a*, such as H7. In this paper, we use only the so-called *A1 notation* 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[5].
- A rectangular range denoted by a cell address and a size, written thus: H4::{3,1}. This *corner-size notation* is not present in typical spreadsheet tools, but turns out to be very convenient for our purposes. The size $\{nr, nc\}$ in braces gives the number of rows and columns in the range, respectively[6]. So H4::{3,1} means precisely the same as H4:H6.

Formulas F are also standard in spreadsheets, and again Figure 1 shows our syntax. A formula can refer to a range using a *range reference* ρ, such as F4 or $G$2. (The tile names t_i in our abstract syntax $\rho^{t_1,\ldots,t_n}$ are automatically deduced annotations that are omitted in concrete examples.) 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 relative/absolute distinction makes a difference *only* during copy/paste; during formula evaluation it is completely ignored.

A range reference in corner-size form can always be expressed in corner-corner form; for example, H$4::{3,1} means precisely the same as H$4:H$6. In the other direction, a corner-corner range reference cannot always be expressed in corner-size form. For example consider B2:B$6; the copy-paste behaviour of this reference is different to, say, B2::{5,1}.

The range on the left-hand side of a range assignment is called a *tile*; tiles 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 between C and D, while its absolute references are unchanged. The advantage of our range-assignment notation, compared to copy/paste in the grid, is that it makes explicit that the entire range shares a single master formula.

3.2 Sheet-defined functions

Next, we extend our Calculation View notation to cover sheet-defined functions (Figure 1). For example, we can abstract the re-usable computational content of the sheet fragment in Section 3.1 as an SDF, like this:

```
function SHOP( F4::{3,1}, G2 ) returns H7 {
  G4::{3,1} = F4 * $G$2
  H4::{3,1} = F4 + G4
  H7 = SUM(H4::{3,1}) }
```

The spreadsheet in Figure 2 contains the body of the SDF as well as two calls to it, the second of which requires elasticity in order to work correctly.

[5] 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; a range G5:G4 is empty, while G6:G4 is simply ill-formed.
[6] Generally, spreadsheets use row,column ordering (e.g. in R1C1 notation, or array indexing) but A1 notation has always been backwards, putting the column first.

In general, an SDF definition $\mathcal{F}$ 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.

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. 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::{3,1}, while the second is placed in G2.

For the sake of simplicity, we only attempt to generalise SDFs that are *tame*, meaning they satisfy the properties:

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 Each range reference in the body of the SDF, evaluated in each cell of the tile in which it appears, 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::{3,1} = 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.

Non-introspective No occurrence of functions like ROW or COLUMN, that implicitly inspect the location of the formula in which they appear.

Non-degenerate Each tile has positive height and width, and each range reference evaluates to a range of positive height and width for at least one cell of the calling tile. These technical restrictions make our proofs simpler by avoiding corner cases, and we know of no useful functions that are thereby excluded.

3.3 Tiles and dependencies

The point of generalisation is to elasticise some of the tiles of the SDF. To do so, it is helpful to name each tile, and to make explicit the tiles on to which each reference points, producing a *labelled SDF* like this:

function SHOP(t_1 F4::{3,1}, t_2 G2) returns H7^{t_5} {
t_3 G4::{3,1} = F4^{t_1} * $G$2^{t_2}
t_4 H4::{3,1} = F4^{t_1} + G4^{t_3}
t_5 H7 = SUM(H4::{3,1}^{t_4}) }

We have given a distinct name, $t_1, t_2, \ldots$ to each tile, including the input tiles. For each reference in the right-hand sides, 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 would read a cell from tile 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.

The *calling tile* of a reference is the tile in whose formula that reference appears; for example, the calling tile of the reference $\$G\$2^{t_2}$ is t_3.

In the SHOP example, each reference has a *unique* 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::{3,2}^{t_3,t_4})

In this (contrived) example, the reference G4::{3,2} covers both tiles t_3 and t_4, and must be so labelled. We show a more realistic example in Section 7.4.

We do not expect that users will write, or even see, these labelled definitions. Rather, labelling the original SDF with a fresh name for each tile, and then computing the unique set of target tiles for each reference, is the first, purely automatic step in our generalisation process.

4 ELASTIC SDFS

The main focus of the paper is the task of *generalising* an SDF $\mathcal{F}$ to an appropriate *elastic SDF* $\widetilde{\mathcal{F}}$. We divide the process into three steps:

(1) *Labelling.* Construct the labelled form $\hat{\mathcal{F}}$ of $\mathcal{F}$ (which is unique up to renaming of tiles), as described in Section 3.3. Because we restrict to static SDFs (Section 3.2), this step is straightforward, and we do not discuss it in detail.
(2) *Generalisation.* Generalise the labelled SDF to an elastic SDF $\widetilde{\mathcal{F}}$. This step is our main technical contribution, and is described in Sections 6 and 7.
(3) *Code generation.* Transform the elastic SDF to executable form, which can be done in a variety of ways (Section 4.4).

4.1 Generalisation by example

We begin with an example to illustrate the process. Suppose we start with the SHOP SDF introduced in Section 3.2. Step 1 is to annotate it with tiles, as described in Section 3.3, thus:

function SHOP(t_1 F4::{3,1}, t_2 G2) returns H7^{t_5} {
t_3 G4::{3,1} = F4^{t_1} * $\$G\$2^{t_2}$
t_4 H4::{3,1} = F4^{t_1} + G4^{t_3}
t_5 H7 = SUM(H4::{3,1}^{t_4}) }

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

function SHOP⟨α⟩(t_1 F4::{α,1}, t_2 G2) returns H7^{t_5} {
t_3 G4::{α,1} = F4^{t_1} * $\$G\$2^{t_2}$
t_4 H4::{α,1} = F4^{t_1} + G4^{t_3}
t_5 H7 = SUM(H4::{α,1}^{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,1}, 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 size to {α+1,1}.) The idea is, of course, that if we instantiate α to 3, we recover exactly the labelled 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 4.3, and how it might be executed in Section 4.4.

4.2 Syntax of elastic SDFs

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 1. We generalize span sizes l, column names N, and row numbers m, to include the possibility of adding a single length variable (enclosed in curly braces).[7]

In the SHOP example, in tile H{4+α} = SUM(H4::{α,1}), the left-hand side shows an elastic row number H{4+α}, and a range reference with an elastic size H4::{α,1}. We elide the curly braces when the coordinate is just a constant, or when it appears as a span size. So we write H4::{α,1}, not H4::{{α},1}.

4.3 Semantics of elastic SDFs

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*.

From a semantic point of view we can easily solve this problem, by using the tile set that labels each reference. 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 the next section, Section 5.

Not every syntactically-correct elastic SDF, as defined in Section 3.2, 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⟨α⟩( t1 F4::{α,1}, t2 G2 ) returns H{4+α} {
  t3 G4::{α,1} = F4^t1 * $G$2^t2
  t4 H4::{3,1} = F4^t1 + G4^t3        /* NB: Bogus! */
  t5 H{4+α} = SUM(H4::{α,1}^t4) }
```

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. For the ambiguous case, consider the tile from the end of Section 3.3:

```
t5 H7 = SUM(G4::{3,2}^t3,t4)
```

[7] 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) make sense with linear combinations. The alert reader will notice that our choice leads to some range references that are grammatical in corner-corner notation but not corner-size notation and vice versa, but these range references will fail to be unambiguous, and thus their existence in the language is unimportant.

When evaluating the reference G4::{3,2}^{t_3,t_4}, for every cell in the range G4::{3,2}, say G6, there should be a unique tile among t_3, t_4 that defines G6 – and there is, namely 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⟨α, β⟩( A1::{α,1} ) returns B1 {
 C1::{β,1} = ...
 B1 = SUM( C1::{β,1} ) }
```

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:

Definition 4.1 (Well-defined elastic SDF). An elastic SDF $\widetilde{\mathcal{F}}$ is well-defined if

(1) It is *unambiguous*, meaning that for every assignment of values to the length variables:
   (a) All tiles have non-negative height and width,
   (b) No two tiles in the target-tile set of the same reference overlap.
   (c) For every labelled range reference $\widetilde{\rho} = \rho^{t_1,\ldots,t_k}$ appearing in a calling tile t_c of $\widetilde{\mathcal{F}}$, and with respect to every cell in t_c, the reference $\widetilde{\rho}$ evaluates to a range of non-negative height and width that is covered by the tiles $t_1, \ldots, t_k$.
(2) It is *determinable*: all its length variables are uniquely determined by the sizes of its arguments.

4.4 Executing elastic SDFs

Our semantics says what a well-defined elastic SDF *means*. We can use this semantics directly as a basis for execution, but doing so requires some extensions to a standard spreadsheet interpreter. An alternative is instead to translate the elastic SDF into a form that is more amenable to direct execution. This translation is not the main focus of the paper, but we sketch three alternatives in Appendix A: using multiple worksheets, using coordinate arithmetic to avoid tile overlaps, or using array-level operations instead of element-level ones. Here, for example, is the second of these alternatives, using overlap avoidance:

```
function SHOP⟨α⟩( F4::{α,1}, G2 ) returns H{4+α} {
 G4::{α,1} = F4 * $G$2
 H4::{α,1} = F4 + G4
 H{4+α} = SUM(H4::{α,1}) }
```

We compute the result in cell H{4+α}, which moves (as α increases) to avoid overlap with the preceding tile.

5 FORMAL SEMANTICS OF SDFS AND SLASTIC 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 $\mathcal{S}$.

A *value* V is either a string or number c, or a 2D array: we write $\{c_{1,1}, \ldots, c_{1,n}; \ldots; c_{m,1}, \ldots, c_{m,n}\}$ for an $[m \times n]$ 2D array, with $m \geq 0$ rows and $n \geq 0$ columns (Figure 3). In common with most

Inelastic address	$a ::= Nm$
Inelastic range	$r ::= a_1 : a_2$
Inelastic reference	$\theta ::= \nu N \mu m$
Inelastic formula	$F ::= (\theta : \theta)^{t_1,\ldots,t_n} \mid c \mid f(F_1, \ldots, F_n)$
Inelastic range assignment	$\mathcal{A} ::= t\ r = F$
Inelastic sheet fragment	$\mathcal{S} ::= \mathcal{A}_1, \ldots, \mathcal{A}_n$
Value	$V ::= c \mid \{c_{1,1}, \ldots, c_{1,n}; \ldots; c_{m,1}, \ldots, c_{m,n}\}$
Binding	$\mathcal{B} ::= t\ r = V$
Context	$\gamma ::= (\mathcal{S}, \mathcal{B}_1 \ldots \mathcal{B}_n)$

Fig. 3. Core syntax for formal semantics

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.[8]

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

The full syntax for our language, given in Figure 1, allows length variables to occur in many forms of syntax. While these variables are needed to express elastic SDFs, they can be eliminated before formula evaluation occurs. If P is a phrase of syntax, let's write $vars(P)$ for the set of variables occurring in P. We say a piece of syntax is *inelastic* when it contains no length variables, that is, $vars(P) = \emptyset$. So, to specify our semantics, we use a simpler core syntax of inelastic formulas as given in Figure 3, which also defines values and other syntax used in this section. As we describe in Section 3.1, any inelastic range $a :: \{\bar{l}_1, \bar{l}_2\}$ or range reference $\theta :: \{\bar{l}_1, \bar{l}_2\}$ in corner-size form can be turned into corner-corner form. Moreover, any singleton range a or range reference θ is equivalent to corner-corner form $a : a$ or $\theta : \theta$. So the core grammar in Figure 3 uses only corner-corner form.

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:

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

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, for example.

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 k, we assume $[\![f]\!]$ is a function from k-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,

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

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 $vN\mu m : v'N\mu'm$ 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*, $\mathcal{B}$, 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. Now, we can complete the definition of context: a *context* γ is a pair $(\mathcal{S}, \mathcal{B}_1 \ldots \mathcal{B}_n)$, where $\mathcal{S}$ is an inelastic sheet fragment, $n \geq 0$, and each $\mathcal{B}_i$ is a binding.

Consider an inelastic (labelled) SDF f:

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

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

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

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

- If $t\ r = F$ is one of the range assignments in $\mathcal{S}$, and address a falls within the range r and F' is the outcome of drag-filling the formula F from the top-left of r to address a, then lookup$(t, a, \gamma) = F'$.
- If $t\ r = c$ is one of the bindings in $\mathcal{B}_1 \ldots \mathcal{B}_k$, and range $r = a : a$, then lookup$(t, a, \gamma) = c$.
- If $t\ r = \{c_{1,1}, \ldots, c_{1,n}; \ldots; c_{m,1}, \ldots, c_{m,n}\}$ is one of the bindings in $\mathcal{B}_1 \ldots \mathcal{B}_k$, 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.

For example, if context γ holds the sheet in the body of the SHOP function from Section 3.3, then lookup$(t_4, H5, \gamma) = F5^{t_1} + G5^{t_3}$.

As explained in Section 4.3, we need the t parameter to uniquely dereference range references in the semantics of elastic SDFs; it is a *key idea* of the semantics.

5.3 Semantics of elastic SDFs

Consider an elastic SDF f:

function $f\langle\alpha_1, \ldots, \alpha_n\rangle(t_1\ r_1, \ldots, t_k\ r_k)$ returns $r^{t'_1,\ldots,t'_j}$ $\{\ \mathcal{S}\ \}$

To give the semantics, we need a substitution operator on syntax, that turns length variables into specific numbers. If $\phi = (\alpha_1 = l_1, \ldots, \alpha_n = l_n)$ is a substitution of actual lengths l_i for length variables, we write $\phi(P)$ for the outcome of substituting the length l_i for each occurrence of length variable α_i in P. If $vars(P) \subseteq \{\alpha_1, \ldots, \alpha_n\}$ we have that $vars(\phi(P)) = \emptyset$, that is, that $\phi(P)$ is inelastic.

The meaning $[\![f]\!]$ of the elastic SDF is the function:

$$\begin{array}{rcl} [\![f]\!] & = & \lambda(V_1,\ldots,V_k).[\![\phi(r)^{t'_1,\ldots,t'_j}]\!](\phi(\mathcal{S}),\mathcal{B}_1\ldots\mathcal{B}_k) \\ & & \text{where each } \mathcal{B}_i = (t_i\ (\phi(r_i)) = V_i) \\ & & \text{for some } \phi = (\alpha_1 = l_1,\ldots,\alpha_n = l_n) \\ & & \text{where size of } \phi(r_i) \text{ equals size of } V_i \text{ for each } i \end{array}$$

In the final constraint, we refer to the *sizes* of inelastic ranges $\phi(r_i)$ 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 rows. The size of an array with m rows and n columns is simply $[m \times n]$, and the size of a constant is $[1 \times 1]$. Our assumption in Section 4.3 that a well-defined elastic SDF is determinable implies that the final constraint uniquely determines the length l_i assigned to each variable α_i by ϕ.

For example, consider our example elastic SDF from Section 1:

function AVERAGE$\langle\alpha\rangle$(t_1 A1:A$\{\alpha\}$) returns B3^{t_4} { $\mathcal{S}$ }

where the sheet fragment $\mathcal{S}$ 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 $[\![$AVERAGE$]\!]$ is the following:

$$\begin{array}{l} \lambda(V_1).[\![\phi(B3)^{t_4}]\!](\phi(\mathcal{S}),\mathcal{B}_1) \\ \text{where } \mathcal{B}_1 = (t_1\ (\phi(A1{:}A\{\alpha\})) = V_1) \\ \text{for some } \phi = (\alpha = l) \\ \text{where size of } \phi(A1{:}A\{\alpha\}) \text{ equals size of } V_1 \end{array}$$

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 $\widetilde{\mathcal{F}}$ to be *well-defined* (that is, both *unambiguous* and *determinable*, see Section 4.3).
(2) We give a generalisation ordering between elastic SDFs, and say what it means for $\widetilde{\mathcal{F}}$ to be a generalisation of a labelled SDF $\hat{\mathcal{F}}$ (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 labelled 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.

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

Elastic SDF generalisation ordering (Section 6.1)

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

Constraint solving (Section 7.2)

Delta variable	$\hat{\alpha}, \hat{\beta}$
Delta constant	$\hat{l} \in \mathbb{Z}$
Constraint	$Q \ ::= \ \hat{\alpha} = \hat{\beta} \mid \hat{\alpha} = 0 \mid \hat{\alpha} + \hat{l}_1 \geq \hat{l}_2$
Intermediate subst	$\theta \ ::= \ \epsilon \mid \theta, \hat{\alpha} \mapsto \hat{\beta} \mid \theta, \hat{\alpha} \mapsto 0$
Delta substitution	$\Theta \ ::= \ \epsilon \mid \Theta, \hat{\alpha} \mapsto 0 \mid \Theta, \hat{\alpha} \mapsto \alpha + \hat{l}$

Fig. 4. Constraints and substitutions

6.1 The generalisation ordering

We start with Step (2). Recall that we have a *labelled SDF* $\hat{\mathcal{F}}$, and we seek its principal generalisation, an *elastic SDF* $\widetilde{\mathcal{F}}$ (see Figure 1). An elastic SDF is still labelled, but it enjoys some length parameters α. So a labelled SDF is just a degenerate 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 the top part of 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.

Definition 6.1 (More general than). An elastic SDF $\widetilde{\mathcal{F}_1}$ is more general than (or, equivalently, a generalisation of) $\widetilde{\mathcal{F}_2}$ if there exists a length substitution that converts $\widetilde{\mathcal{F}_1}$ to $\widetilde{\mathcal{F}_2}$ (ignoring the length variable declarations themselves).

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

Proposition 6.2. *If $\widetilde{\mathcal{F}}$ is a well-defined generalisation of $\hat{\mathcal{F}}$, then $\widetilde{\mathcal{F}}$ is semantically equivalent to $\hat{\mathcal{F}}$ on inputs of the original size.*

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.

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. For example:

```
function ND0(t1 A1::{3,1}, t2 A5::{3,1}) returns A13^t4 {
 t3  A9::{3,1} = 1
 t4  A13 = SUM(A9::{3,1}^t3) } /* 3 */
```

This SDF has the following well-defined generalisations:

```
function ND1⟨α, β⟩(t1 A1::{α,1}, t2 A5::{β,1}) returns A13^t4 {
 t3  A9::{3,1} = 1
 t4  A13 = SUM(A9::{3,1}^t3) } /* Always 3 */
```

```
function ND2⟨α, β⟩(t1 A1::{α,1}, t2 A5::{β,1}) returns A13^t4 {
 t3  A9::{α,1} = 1
 t4  A13 = SUM(A9::{α,1}^t3) }  /* Equal to the length of the first input */
function ND3⟨α, β⟩(t1 A1::{α,1}, t2 A5::{β,1}) returns A13^t4 {
 t3  A9::{β,1} = 1
 t4  A13 = SUM(A9::{β,1}^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? 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⟨α, β, γ⟩(t1 A1::{α,1}, t2 A5::{β,1}) returns A13^t4 {
 t3 A9::{γ,1} = 1
 t4 A13 = SUM(A9::{γ,1}^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 simple and we claim that the cases in which it is needed are contrived. Still, there are some examples that are arguably not contrived, such as the following which may be an attempt to count the number of elements of the input:

```
function MYCOUNT0(t1 A1::{3,1}) returns C1^t3 {
 t2  B1::{3,1} = 1
 t3  C1 = SUM(B1::{3,1}^t2) }
```

A design change that would give the desired result in this scenario (but does not address the ND0 case above) would be to require that adjacent tiles that have the same length in the original SDF have the same length variable. We are undecided on the merits of this change and simply note that the results of the rest of the paper hold whether or not it is made (in Definition 6.3 and step 3 of the algorithm at the beginning of Section 7).

6.3 Problem 2: arbitrary 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⟨α⟩( t1 F4::{α,1}, t2 G2 ) returns {E+α}7^t5 {
 t3 G4::{α,1} = F4^t1 * $G$2^t2
 t4 H4::{α,1} = F4^t1 + G4^t3
 t5 {E+α}7 = SUM(H4::{α,1}^t4) }
```

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 stop this nonsense is to require that the upper-left corner of each tile of the elastic SDF be constant; that is, mention no length variables.[9] Tile t_5 above violates this because its top-left corner is at {E+α}7.

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

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⟨α⟩(t1 A1::{α,1}) returns B1^t2 {
  t2 B1 = COUNT(A1::{α,1}^t1) }  /* Returns length of the input */
function G⟨α⟩(t1 A1::{α,1}) returns B1::{α,1}^t2 {
  t2 B1::{α,1} = 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.

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::{2,1}) returns B1::{2,1}^t2 { t2 B1::{2,1} = A1^t1 }
```

has the following incomparable generalisations[10]:

```
function F⟨α⟩(t1 A1::{α,1}) returns B1::{α,1}^t2 {
  t2 B1::{α,1} = A1^t1 }  /* Returns the entire input */
function F⟨α⟩(t1 A1::{α + 2,1}) returns B1::{2,1}^t2 {
  t2 B1::{2,1} = A1^t1 }  /* Returns first two elements of input */
function F⟨α⟩(t1 A1::{α + 2,1}) returns B1::{2,1}^t2 {
  t2 B1::{2,1} = A{α+1}^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 saying (in Definition 6.4 below) that every reference should be *well-behaved*, and exploring various possible definitions for "well-behaved".

6.6 Regularity

It is no good choosing at random among incomparable generalisations. 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 next two problems (Sections 6.3 and 6.4) by restricting the set of generalisations among which we choose, to the *semi-regular* ones (defined shortly).
- We solve the final problem (Section 6.5) by further restricting the generalisations we consider to those in which every range reference is *well-behaved*. The definition of "well-behaved" will somehow express common computational patterns in a predictable way. Rather than define well-behavedness once and for all here, we instead *parameterise* our generalisation

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

algorithm, and its proof of principality, over this choice. This enables us to explore a variety of choices for well-behavedness: we present two in this paper, but others are possible.

The following definitions make the above outline precise.

Definition 6.3 (Semi-regular generalisation). An elastic SDF $\widetilde{\mathcal{F}}$ is a *semi-regular generalisation* of a labelled SDF $\hat{\mathcal{F}}$ if it satisfies the following conditions:

(1) $\widetilde{\mathcal{F}}$ is a generalisation of $\hat{\mathcal{F}}$.
(2) Every tile of $\widetilde{\mathcal{F}}$ has non-negative height and width for every assignment of values to the length variables.
(3) The upper-left corner of each tile of $\widetilde{\mathcal{F}}$ is constant (Section 6.3).
(4) Every tile of non-constant height in $\widetilde{\mathcal{F}}$ has height at least 2 in $\hat{\mathcal{F}}$, and likewise for the width (Section 6.4).

Definition 6.4 (Regular generalisation). An elastic SDF $\widetilde{\mathcal{F}}$ is a *regular generalisation* of a labelled SDF $\hat{\mathcal{F}}$ if it is a semi-regular generalisation of $\hat{\mathcal{F}}$ *and* each range reference $\widetilde{\rho}$ in $\widetilde{\mathcal{F}}$ is *well-behaved* (Section 6.5).

Definition 6.5 (Principal regular generalisation). An elastic SDF $\widetilde{\mathcal{F}}^*$ is the *principal* regular generalisation of $\hat{\mathcal{F}}$ if it is a regular generalisation of $\hat{\mathcal{F}}$ and is more general than every other regular generalisation of $\hat{\mathcal{F}}$.[11]

7 ELASTICITY INFERENCE

Next, we turn our attention to the task of finding the principal regular generalisation of an SDF. Our approach is quite conventional: first generate constraints, and then find their principal solution. Obviously, some of the constraints depend on the definition of well-behaved references. Thus we define:

Definition 7.1 (Generalisation system). A *generalisation system* consists of:

- A class of *supported SDFs*, a subset of labelled SDFs as defined in Sections 3.2 and 3.3;
- A predicate for *well-behaved range references* in supported SDFs; and
- A *constraint generator* that takes a range reference in the "master" elastic SDF $\widetilde{\mathcal{F}}_0$ (defined below) generated from a supported SDF and returns a set of constraints.

Given a generalisation system, the elasticity inference algorithm is as follows:

(1) Convert all tile ranges of $\hat{\mathcal{F}}$ to corner-size notation and all range references to corner-corner notation (expanding single cell references to pairs of identical corners).
(2) Generate a "master" elastic version $\widetilde{\mathcal{F}}_0$ of $\hat{\mathcal{F}}$ by adding a fresh *delta variable* $\hat{\alpha}$ to the height and width of each tile, and to each row or column reference. Setting all the delta variables to zero recovers the original function $\hat{\mathcal{F}}$. Unlike length variables, delta variables can potentially take negative values. (Thus $\widetilde{\mathcal{F}}_0$ is not truly an elastic SDF.)
(3) Generate a set of *constraints* on the delta variables, in the syntax given in Figure 4, as follows:
   (a) For each tile, if the height was 1 in $\hat{\mathcal{F}}$, then constrain the height delta variable equal to 0[12]; otherwise constrain the height to be non-negative. Do likewise with the width.

[11] The principal regular generalisation will only ever be unique up to renaming of length variables and addition and removal of unused length variables (because regularity does not require determinability), but we ignore these technicalities and refer to it as if it were unique.

[12] One might ask, why add a height delta variable only to immediately constrain it to zero? Ensuring that every tile has a height delta variable makes the system-specific constraint generators slightly easier to state, e.g., in step 1 in Section B.2.

(b) Call the generalisation system's constraint generator on each range reference $\widetilde{\rho}$ in $\widetilde{\mathcal{F}_0}$.

(4) Find the principal solution of the constraints, a delta substitution Θ^* that maps every delta variable either to zero or to $\alpha + \hat{l}$ where α is a length variable (Section 7.2).

(5) Apply Θ^* to $\widetilde{\mathcal{F}_0}$ to produce the principal regular generalisation $\widetilde{\mathcal{F}}^*$ of $\hat{\mathcal{F}}$.

7.1 Elasticity inference by example

We illustrate this sequence of steps using the SHOP SDF introduced in Section 3.2 and the simplified generalisation system of Appendix B:

(1) $\hat{\mathcal{F}}$ after the initial conversions:

```
function SHOP( t1 F4::{3,1}, t2 G2::{1,1}) returns H7:H7^t5 {
  t3 G4::{3,1} = F4:F4^t1 * $G$2:$G$2^t2
  t4 H4::{3,1} = F4:F4^t1 + G4:G4^t3
  t5 H7::{1,1} = SUM(H4:H6^t4) }
```

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

```
function SHOP( t1 F4::{3+α̂1,1}, t2 G2::{1+α̂2,1})
   returns H{7+α̂3}:H{7+α̂4}^t5 {
  t3 G4::{3+α̂5,1} = F{4+α̂6}:F{4+α̂7}^t1 * $G${2+α̂8}:$G${2+α̂9}^t2
  t4 H4::{3+α̂10,1} = F{4+α̂11}:F{4+α̂12}^t1 + G{4+α̂13}:G{4+α̂14}^t3
  t5 H7::{1+α̂15,1} = SUM(H{4+α̂16}:H{6+α̂17}^t4) }
```

Note that we elasticise only the *size* of each tile (on the LHS), not its *position*, to respect item 3 of Definition 6.3.

(3) Generate constraints. Here we show $\widetilde{\mathcal{F}_0}$ again, with each constraint attached to the part of $\widetilde{\mathcal{F}_0}$ it was generated from and marked with the step of the constraint generation procedure (Appendix B.2) that generated it. Unmarked constraints are from step 3a of the main algorithm.

```
function SHOP( t1 F4::{3+α̂1,1}[3+α̂1≥0], t2 G2::{1+α̂2,1}[α̂2=0])
   returns H{7+α̂3}:H{7+α̂4}^t5[(c) α̂15=α̂3=α̂4=0] {
  t3 G4::{3+α̂5,1}[3+α̂5≥0]
     = F{4+α̂6}:F{4+α̂7}^t1[(a) α̂1=α̂5, α̂6=α̂7=0]
       * $G${2+α̂8}:$G${2+α̂9}^t2[(c) α̂2=α̂8=α̂9=0]
  t4 H4::{3+α̂10,1}[3+α̂10≥0]
     = F{4+α̂11}:F{4+α̂12}^t1[(a) α̂1=α̂10, α̂11=α̂12=0]
       + G{4+α̂13}:G{4+α̂14}^t3[(a) α̂5=α̂10, α̂13=α̂14=0]
  t5 H7::{1+α̂15,1}[α̂15=0]
     = SUM(H{4+α̂16}:H{6+α̂17}^t4[(b) α̂16=0, α̂17=α̂10]) }
```

(4) Solving the constraints yields the delta substitution $\{\hat{\alpha}_1, \hat{\alpha}_5, \hat{\alpha}_{10}, \hat{\alpha}_{17}\} \mapsto \alpha - 3$, and all other delta variables are zero.

(5) The result of applying this substitution to $\widetilde{\mathcal{F}_0}$ 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) substitution that solves the constraints. The syntax of constraints and substitutions is given in Figure 4.

The solution to a set of constraints is a *delta substitution*, which maps each delta variable to zero (meaning that the coordinate is inelastic) or to $\alpha + \hat{l}$ (meaning that the coordinate has elastic variable α); see Figure 4. A delta substitution Θ *satisfies* a constraint if applying Θ to both sides of the constraint makes the constraint true, remembering that length variables α are non-negative. For example $\hat{\alpha} \mapsto \alpha + 3$ satisfies $\hat{\alpha} \geq 3$.

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

The generalisation systems that we consider will have the property that every delta variable has a lower-bound constraint, 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 4).

7.3 Proof of principality

In this section we prove that our elasticity inference algorithm indeed finds the principal regular generalisation of a labelled SDF $\hat{\mathcal{F}}$. Here is the key theorem:

Theorem 7.2 (Principal regular generalisation). *In a sound generalisation system, every supported SDF $\hat{\mathcal{F}}$ has a principal regular generalisation $\widetilde{\mathcal{F}}^*$, $\widetilde{\mathcal{F}}^*$ is unambiguous, and the algorithm given at the start of Section 7 finds it.*

Proof. See Appendix C. □

Since the constraint generation algorithm and the definition of regularity are both parameterised over the generalisation system, the theorem is predicated on the *soundness* of the generalisation system (see Definition 7.1). Soundness captures the properties that the generalisation system must have to prove the principality theorem.

Definition 7.3 (Sound generalisation system). A generalisation system is *sound* if the following properties hold for every supported SDF $\hat{\mathcal{F}}$ with "master" elastic SDF $\widetilde{\mathcal{F}}_0$:

(1) Every regular generalisation of $\hat{\mathcal{F}}$ is unambiguous (Definition 4.1).
(2) No constraint requires $\hat{\alpha} > 0$, for any delta-variable $\hat{\alpha}$.
(3) For every range reference $\widetilde{\rho}$ in $\widetilde{\mathcal{F}}_0$ and for each row or column reference, denoted by $\overline{\chi}$, in $\widetilde{\rho}$, the constraints generated for $\widetilde{\rho}$ include an equality constraint of the delta variable of the row or column reference $\overline{\chi}$ to either 0 or a height or width delta variable of a tile.
(4) For every delta substitution Θ', if the elastic SDF $\widetilde{\mathcal{F}}' = \Theta'(\widetilde{\mathcal{F}}_0)$ is a semi-regular generalisation of $\hat{\mathcal{F}}$, then for each reference $\widetilde{\rho}$ in $\widetilde{\mathcal{F}}_0$, Θ' satisfies the constraints generated for $\widetilde{\rho}$ if and only if the reference $\widetilde{\rho}'$ corresponding to $\widetilde{\rho}$ in $\widetilde{\mathcal{F}}'$ is well-behaved.

Property 1 is required because we are only interested in unambiguous generalisations. Property 2 ensures that the constraints allow setting all the delta-variables to zero; we need that possibility to guarantee that the solution $\widetilde{\mathcal{F}}^*$ is actually a generalisation of $\hat{\mathcal{F}}$. Together with Step 3(a) of the inference algorithm in Section 7, Property 3 ensures that every delta-variable is either constrained

to zero, or has a non-positive lower bound. This property of the generated constraints is required by step (3) the constraint solver (Section 7.2), which eliminates the delta-variables.

Finally, property 4 says that the constraint generator precisely characterises the well-behavedness of references. Observe that in step 2 of the elasticity inference algorithm we added enough delta variables that *any* semi-regular generalisation $\widetilde{\mathcal{F}}'$ of $\hat{\mathcal{F}}$ is the result of applying *some* delta substitution Θ' to $\widetilde{\mathcal{F}}_0$, so that $\Theta'(\widetilde{\mathcal{F}}_0) = \widetilde{\mathcal{F}}'$. Why? Because the only parts of $\widetilde{\mathcal{F}}$ that can differ from $\hat{\mathcal{F}}$ are tile sizes and row and column references (not tile upper-left corners by semi-regularity condition 3). Property 4 then implies that Θ' satisfies the reference well-behavedness constraints if and only if $\widetilde{\mathcal{F}}'$ is regular.

Proposition 7.4. *In a sound generalisation system, every supported SDF $\hat{\mathcal{F}}$ is a regular generalisation of itself.*

Proof. See Appendix C. □

What this proposition means is that we can expect a generalisation system to degrade gracefully: the elasticity algorithm cannot fail to produce a principal generalisation. At worst it will make inelastic some part that one might hope would be generalised, with the worst case being $\widetilde{\mathcal{F}}^* = \hat{\mathcal{F}}$ itself.

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

7.4 The generalisation system

Our elasticity algorithm is parameterised over the *generalisation system* (Definition 7.1). Principality is guaranteed (by Theorem 7.2) for any generalisation system that is sound (Definition 7.3).

We have studied two generalisation systems. Space precludes giving the details here, but to summarise:

- The simple generalisation system (Appendix B) handles SDFs in which each reference has only one target tile. This is enough for the SHOP example.
- The full generalisation system (Appendix D) handles a richer class of SDFs, but in exchange it is more complicated.

The extra expressiveness of the full system is important in practice. For example, here is a function that computes the post-transaction balances of a bank account with interest compounded daily:

```
/* 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::{7,2} )
  returns F4::{7,1}^t4 {
 t1 C4::{7,1} = A4^tx − A3^ts,tx   /* Interval between transactions */
 t2 D4::{7,1} = POWER(1+$F$1^tr, C4^t1)   /* Interest multiplier */
 t3 E4::{7,1} = F3^to,t4 * D4^t2   /* New balance after interest */
 t4 F4::{7,1} = E4^t3 + B4^tx  }   /* 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 computed in cell C4, the reference A3 points to the start date input tile t_s; but when computed in cell C5, the A3 has become A4 (via copy/paste), and hence points to the

date on the first transaction, in tile t_x. 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, our full generalisation system, described in Appendix D, handles arbitrary tame SDFs, and still enjoys principal generalisations.

8 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 array-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[13].

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-as-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 the elasticisation process is a substantial undertaking that is part of our future work.

8.1 Prototype of SDFs for User Study

We adapted an existing research prototype of sheet-defined functions written as an add-in for Microsoft 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 Appendix A.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 Appendix A.3.

8.2 Participants and tasks

We had twenty participants (seven female) aged 18-35 (mean 24), an adequate sample for preliminary field research in human-computer interaction [Caine 2016].

[13] https://support.office.com/en-us/article/Dynamic-arrays-and-spilled-array-behavior-205c6b06-03ba-4151-89a1-87a7eb36e531; https://support.google.com/docs/answer/3093275?hl=en; last accessed: March 6, 2019

	B	C	D	E
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. 5. A spreadsheet containing the definition of the EXPENSES SDF.

We developed tasks that were representative of real-world spreadsheet tasks, to maintain external validity. 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.

For example, here is a description of a task based on 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 7 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 (recall that this notation is not user-facing; the tiles and variables are automatically inferred).

```
function EXPENSES( t1 C6::{α,1}, t2 D6::{α,1} ) returns E17^t4 {
  t3 E6::{α,1} = IF(D6^t2 < C6^t1, D6^t2, C6^t1)
  t4 E17 = SUM(D6::{α,1}^t2) − SUM(E6::{α,1}^t3)
}
```

This example needs only the simplified generalization system of Section B. The array SDF is similar and written as:

```
function ARRAY_EXPENSES( C6, D6 ) returns E17 {
  E6 = IF(D6<C6, D6, C6)
  E17 = SUM(D6) − SUM(E6)
}
```

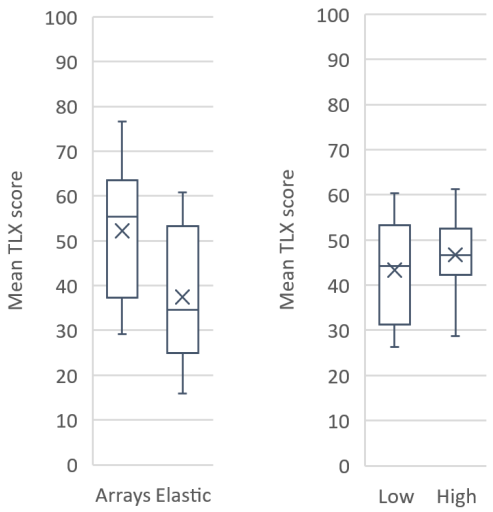
Appendix E includes additional details of our user study tasks and participants.

8.3 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) on 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.

8.4 Results

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. 6. 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 6a) (M = 37.46, SD = 14.24) than arrays (M = 52.25, SD = 14.80), $F(1, 18)$[14] = 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), $F(1,18) = 1.93$, $p = 0.2$. This implies that there was no ordering effect of SDF version on cognitive load. 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. However, the effect of SDF version on workload was slightly greater if users started the study with elastic SDFs, suggesting that participants perceived a larger difference in workload between arrays and elastic SDFs if they started with elastic SDFs.

Our prototype was built using the Office Add-ins Platform,[15] 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). During the study, we observed that this large improvement in the time required with elastic SDFs can be attributed simply to the fact that participants spent longer to formulate logic in terms of array combinators.

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 and/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 6b). There was also no interaction effect between SDF version or programming experience, $F(1,18) = 0.09$, $p = 0.77$, implying that the observed differences in cognitive load between elastic SDFs vs array programming were not affected by users' programming experience. This suggests that elastic SDFs offer similar cognitive benefits to novice / low-expertise as well as high-expertise programmers.

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. 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 populated for all values held in that 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.

[14]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 [Field 2013].
[15]https://docs.microsoft.com/en-us/office/dev/add-ins/overview/office-add-ins, last accessed: March 6, 2019

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. 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 single formula to a range, but preferred elastic SDFs for calculations involving complex array combinators, as well as because it allowed the use of familiar range notation.
- 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.

9 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. [[n. d.]] 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's comprehensive monograph on spreadsheet technology [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 formulae 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 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. Our semantics does not attempt to capture the details of incremental recalc of spreadsheets. A recent work by Mokhov et al. [2018] explores the connection between incremental spreadsheet recalc and build systems.

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

Acknowledgements. Suppressed for anonymous submission.

REFERENCES

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

Laura Beckwith, Derek Inman, Kyle Rector, and Margaret Burnett. 2007. On to the real world: Gender and self-efficacy in Excel. In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*. IEEE, 119–126.

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

Alan F Blackwell, Margaret M Burnett, and Simon Peyton Jones. 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*. IEEE, 47–54.

Robert P Bostrom, Lorne Olfman, and Maung K Sein. 1990. The Importance of Learning Style in End-User Training. *MIS Quarterly* 14, 1 (mar 1990), 101. https://doi.org/10.2307/249313

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

Martin Erwig, Robin Abraham, Steve Kollmansberger, and Irene Cooperstein. 2006. Gencel: a program generator for correct spreadsheets. *Journal of Functional Programming* 16, 3 (2006), 293–325.

Andy Field. 2013. *Discovering Statistics Using IBM SPSS Statistics* (4th ed.). Sage Publications Ltd.

Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131–174.

S. Gulwani, O. Polozov, and R Singh. [n. d.]. Program synthesis. *Foundations and Trends in Programmign Languages* 4 ([n. d.]).

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

Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. 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)*. ACM, New York, NY, USA, 112–127. https://doi.org/10.1145/2986012.2986018

Jorge Mendes and João Saraiva. 2017. Tabula: A Language to Model Spreadsheet Tables. *CoRR* abs/1707.02833 (2017). arXiv:1707.02833 http://arxiv.org/abs/1707.02833

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *PACMPL* 2, ICFP (2018), 79:1–79:29. https://doi.org/10.1145/3236774

Jocelyn Paine. 2008. Ensuring spreadsheet integrity with model master. *arXiv preprint arXiv:0801.3690* (2008).

Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. 2003. A User-centred Approach to Functions in Excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/944705.944721

Advait Sarkar, Andrew D. Gordon, Neil Toronto, and Simon Peyton Jones. 2018. Calculation View: multiple-representation editing in spreadsheets. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 85–93. https://doi.org/10.1109/VLHCC.2018.8506584

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

Peter Sestoft and Jens Zeilund Sørensen. 2013. Sheet-defined functions: implementation and initial evaluation. In *International Symposium on End User Development*. Springer, 88–103.

A TRANSLATION OF ELASTIC SDFS TO EXECUTABLE FORM

In Section 4.3 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 several alternative routes for execution. A detailed description and evaluation of these implementation strategies is beyond the scope of this paper. 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 A.1 (Overlap-free elastic SDF). An elastic SDF $\widetilde{\mathcal{F}}$ is *overlap-free* if no two of its tiles overlap for any length variable assignment.

A.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⟨α⟩( S1!F4::{α,1}, S2!G2 ) returns S5!H7 {
  S3!G4::{α,1} = S1!F4 * S2!$G$2
  S4!H4::{α,1} = S1!F4 + S3!G4
  S5!H7 = SUM(S4!H4::{α,1})   /* No overlap because tiles are
                                 on separate sheets S4, S5 */
}
```

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

A.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⟨α⟩( F4::{α,1}, G2 ) returns H{4+α} {
  G4::{α,1} = F4 * $G$2
  H4::{α,1} = F4 + G4
  H{4+α} = SUM(H4::{α,1})   /* 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 F. 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 1. The specific algorithm we choose is rather naive but gives adequate results at least on simple examples.

A.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 4.3 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 $F3^{t_o,t_4}$ 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.

B A SIMPLIFIED GENERALISATION SYSTEM

In this section, we present a simplified SDF generalisation system with just enough features to handle the SHOP example. The simplified system supports all *basic* SDFs, defined as follows:

Definition B.1 (Basic SDF). A tame SDF is *basic* if it has the property that, once it is correctly labelled, the target-tile set on each reference is a singleton.

In the simplified system, well-behaved references are restricted to three simple, common kinds (defined in Section B.1) that just suffice for the SHOP example. In contrast, the full system, described in Appendix D, supports *all* tame SDFs (not only basic ones) and allows more kinds of references. Every generalisation of a basic SDF that is regular according to the simplified system is regular according to the full system, but not vice versa. Thus, for certain basic SDFs, while the simplified system produces a generalisation that is principal according to its own definition of regularity, the full system produces a more general elastic SDF.

B.1 Well-behaved references

In Section 6.5 we saw the need for a well-behavedness condition that entails unambiguity, captures common patterns of computation, and excludes other possible incomparable generalisations. This section defines that condition.

First we need to introduce some new concepts to analyse the structure of references. Although a range is normally given by a pair of corners and columns are normally denoted by letters, it's more useful for us to think of it as an intersection of a row *span* and a column span, each with two endpoints that are numeric *axis positions* (m in Figure 1), so that we can treat the row and column axes symmetrically. For example, the range F4:G6 is the intersection of the row span 4:6 and the column span 6:7. Likewise, a range reference is an intersection of a row *span reference* and a column span reference, each of which has two *axis position references* that may be absolute or relative.

Below, we recall the abstract syntax for an elastic axis position from Figure 1, and define the syntax for an elastic axis position reference:

Axis position	$m \in \mathbb{Z}^+$
Elastic axis position	$\overline{m} ::= m \mid \{m + \alpha\}$
Elastic axis position reference	$\overline{\chi} ::= \mu\overline{m}$

An (inelastic) axis position reference χ is an elastic axis position where in fact $vars(\chi) = \emptyset$, so that χ has form μm.

By the *coordinate* of an axis position reference, we mean the underlying number (or elastic expression) without regard to whether the reference is absolute or relative.

Since each range reference in a basic SDF has only one target tile, for a generalisation to be unambiguous, we just need to ensure that its row span reference stays inside the row span of the target tile and the endpoints do not flip to produce a negative height, and likewise for the column span reference. To make it as obvious as possible that these properties hold in the simplified system, we stick to three simple kinds of span references that appear in the SHOP example. In the full system, showing that well-behavedness implies unambiguity requires a bit more case analysis.

Definition B.2 (Stable and superstable references). A row reference $\overline{\chi}$ appearing in the formula of a calling tile t_c in an elastic SDF is *stable* if $\overline{\chi}$ is absolute, or t_c is vertically inelastic, or both; it is *superstable* if it is absolute or the height of t_c is 1. The analogous definitions apply to column references.

Intuitively:

- A stable row reference points to a fixed number of different target rows, independent of the values of any length variables, when evaluated across all caller rows.
- A superstable row reference points to a single target row.
- An unstable row reference points to an arbitrary number of target rows depending on the (variable) height of t_c.

Definition B.3 (Well-behaved reference in the simplified system). A range reference $\widetilde{\rho}$ appearing in a basic elastic SDF $\widetilde{\mathcal{F}}$ is *well-behaved* if it satisfies the following conditions:

(1) Let t_c be the calling tile and t_t be the (single) target tile of $\widetilde{\rho}$. Let $\overline{\chi}_1 : \overline{\chi}_2$ be the row span reference of $\widetilde{\rho}$. Then $\overline{\chi}_1 : \overline{\chi}_2$ is of one of the following three kinds:
   - *Inelastic*: The height of t_t is a constant, $\overline{\chi}_1$ and $\overline{\chi}_2$ are both stable, and their coordinates are both constant.
   - *Lockstep*: The heights of t_c and t_t are the same non-constant expression, $\overline{\chi}_1 = \overline{\chi}_2$, $\overline{\chi}_1$ is relative, and $\overline{\chi}_1$ points to the top row of t_t when evaluated at the top row of t_c. (By "points to", we mean for all values of length variables, not just in $\hat{\mathcal{F}}$. In this case, that means the coordinate of $\overline{\chi}_1$ must be constant.)
   - *Whole*: The height of t_t is non-constant, $\overline{\chi}_1$ and $\overline{\chi}_2$ are superstable, $\overline{\chi}_1$ points to the top row of t_t, and $\overline{\chi}_2$ points to the bottom row of t_t.

(2) The analogous condition for the column span reference of $\widetilde{\rho}$.

Examples of all three kinds of span references appear in the elastic version of SHOP shown in Section 4.1:

- All the column span references are of the Inelastic kind, as are the row span references that are part of the references $G$2 in the formula for t_3 and H7 in the returns clause. The $2 is stable because it is absolute, and the H7 in the returns clause is implicitly treated as absolute, while all the other references that are endpoints of Inelastic spans are stable because the calling tile has size 1 on the relevant axis.
- The row span reference in SUM(H4::{α,1}), which is SUM(H4:H{3+α}) in corner-corner notation, is of the Whole kind; the row references are superstable because the calling tile t_5 has height 1.
- The remaining row span references are of the Lockstep kind.

B.2 Constraint generation

Given a span reference $\overline{\sigma}$ in the "master" elastic SDF $\widetilde{\mathcal{F}_0}$, how can we generate a *conjunction* of constraints on a delta substitution Θ that is equivalent to the *disjunction* of $\Theta(\overline{\sigma})$ being of the allowed kinds of span references (Inelastic, Lockstep, or Whole)? The secret is that we can rule out at least one of Lockstep and Whole just by looking at $\widetilde{\mathcal{F}_0}$: Whole requires that the endpoints of $\overline{\sigma}$ be superstable, while Lockstep requires that they not be. Furthermore, the kinds are such that if we rule out Whole but not Lockstep, we're able to generate a set of constraints equivalent to $\Theta(\overline{\sigma})$ being Lockstep *or* Inelastic; likewise, if we rule out Lockstep but not Whole, we can generate a set of constraints equivalent to $\Theta(\overline{\sigma})$ being Whole *or* Inelastic. (If we rule out both Lockstep and Whole, it's easy to generate constraints equivalent to $\Theta(\overline{\sigma})$ being Inelastic.) Thus, based on $\widetilde{\mathcal{F}_0}$, we can always generate a set of constraints equivalent to the disjunction of the kinds that we haven't ruled out.

Based on this insight, the constraint generation procedure for a range reference $\widetilde{\rho}$ in $\widetilde{\mathcal{F}_0}$ is as follows:

(1) Let t_t and t_c be the target tile and calling tile of $\widetilde{\rho}$, let $\overline{\chi}_1 : \overline{\chi}_2$ be its row span reference, let $\chi_1 : \chi_2$ be the original row span reference in $\hat{\mathcal{F}}$, let $\hat{\alpha}_t$ and $\hat{\alpha}_c$ be the height delta variables of t_t and t_c, let $\hat{\alpha}_1$ and $\hat{\alpha}_2$ be the delta variables of $\overline{\chi}_1$ and $\overline{\chi}_2$, and try the following cases in order:
   (a) *(Lockstep)* If $\chi_1 = \chi_2$, χ_1 is relative, and t_t and t_c have the same initial height (at least 2), then we anticipate this span reference is Lockstep, but it may turn out to be Inelastic. Constrain $\hat{\alpha}_t = \hat{\alpha}_c$ and $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$.
   (b) *(Whole)* Otherwise, if the initial height of t_t is at least 2, χ_1 and χ_2 are both superstable, χ_1 points to the top row of t_t in $\hat{\mathcal{F}}$, and χ_2 points to the bottom row of t_t in $\hat{\mathcal{F}}$, then we anticipate this span reference 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, this span reference is Inelastic. Constrain $\hat{\alpha}_t = \hat{\alpha}_1 = \hat{\alpha}_2 = 0$. In addition, if either χ_1 or χ_2 is relative, then constrain $\hat{\alpha}_c = 0$ to ensure that $\overline{\chi}_1$ and $\overline{\chi}_2$ will be stable.

(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 3).

B.3 Soundness of the simplified system

We can now prove:

Theorem B.4. *The simplified generalisation system is sound.*

Proof. An exercise in case analysis. More detail is in Appendix C. □

Corollary B.5. *In the simplified generalisation system, every basic SDF $\hat{\mathcal{F}}$ has a principal regular generalisation $\widetilde{\mathcal{F}}^*$, $\widetilde{\mathcal{F}}^*$ is unambiguous, and the algorithm of Section 7 finds $\widetilde{\mathcal{F}}^*$.*

An interesting property of the simplified system (which does not hold in the full system) is the following:

Proposition B.6. *In the principal regular generalisation $\widetilde{\mathcal{F}}^*$ of a basic SDF $\hat{\mathcal{F}}$, the height and width of every tile is either a constant or a length variable, not a length variable plus a constant.*

Proof. See Appendix C. □

C PROOFS DEFERRED FROM THE MAIN TEXT

Theorem 7.2 (Principal regular generalisation). *In a sound generalisation system, every supported SDF $\hat{\mathcal{F}}$ has a principal regular generalisation $\widetilde{\mathcal{F}}^*$, $\widetilde{\mathcal{F}}^*$ is unambiguous, and the algorithm given at the start of Section 7 finds it.*

Proof. First we need to show that every delta variable that survives step (1) of constraint solving has a lower bound, as required by the constraint solving algorithm. This is true for tile height and width delta variables by step 3a of the high-level algorithm, and it is true for row and column reference delta variables by soundness property 3.

Let $\widetilde{\mathcal{F}}^* = \Theta^*(\widetilde{\mathcal{F}}_0)$ be the elastic SDF found by the algorithm. Since Θ^* satisfies the tile size constraints, it's easy to show that $\widetilde{\mathcal{F}}^*$ satisfies conditions (2)–(4) for semi-regularity. Next we claim that $\widetilde{\mathcal{F}}^*$ is a generalisation of $\hat{\mathcal{F}}$. It is clear that $\widetilde{\mathcal{F}}_0$ can be specialised to $\hat{\mathcal{F}}$ by setting all delta variables to 0. This is still true of $\theta(\widetilde{\mathcal{F}}_0)$ after we eliminate the equality constraints. All lower bounds placed on delta variables are nonpositive, both in the tile size constraints and in the reference regularity constraints by soundness property 2, so each delta variable $\hat{\alpha}$ that remains at this point has a nonpositive lower bound $\hat{l}$ and is replaced by $\alpha + \hat{l}$, with a different α for each such $\hat{\alpha}$. So the length substitution that maps each α to the corresponding $-\hat{l}$ (which is non-negative) specialises $\widetilde{\mathcal{F}}^*$ to $\hat{\mathcal{F}}$, which completes the proof that $\widetilde{\mathcal{F}}^*$ is a semi-regular generalisation of $\hat{\mathcal{F}}$. Since Θ^* satisfies the constraints generated for every reference, every reference in $\widetilde{\mathcal{F}}^*$ is well-behaved by soundness property 4. Thus $\widetilde{\mathcal{F}}^*$ is a regular generalisation of $\hat{\mathcal{F}}$.

To show that $\widetilde{\mathcal{F}}^*$ is the principal regular generalisation of $\hat{\mathcal{F}}$, it remains to show that it is more general than every other regular generalisation $\widetilde{\mathcal{F}}'$ of $\hat{\mathcal{F}}$. As observed above, $\widetilde{\mathcal{F}}' = \Theta'(\widetilde{\mathcal{F}}_0)$ for some delta substitution Θ'. Since $\widetilde{\mathcal{F}}'$ satisfies conditions (2)–(4) for semi-regularity, it's easy to show that Θ' satisfies the tile size constraints. Since all references in $\widetilde{\mathcal{F}}'$ are well-behaved, Θ' satisfies the reference regularity constraints by soundness property 4. Thus Θ' satisfies all the constraints. Since Θ^* is the most general solution to the constraints, there exists a length substitution ϕ such that $\Theta' = \phi \circ \Theta^*$. Then we have $\phi(\widetilde{\mathcal{F}}^*) = \phi(\Theta^*(\widetilde{\mathcal{F}}_0)) = \Theta'(\widetilde{\mathcal{F}}_0) = \widetilde{\mathcal{F}}'$, so $\widetilde{\mathcal{F}}^*$ is more general than $\widetilde{\mathcal{F}}'$.

Finally, by soundness property 1, $\widetilde{\mathcal{F}}^*$ is unambiguous. □

Proposition 7.4. *In a sound generalisation system, every supported SDF $\hat{\mathcal{F}}$ is a regular generalisation of itself.*

Proof. As we observed in the proof of Theorem 7.2, all lower bounds on delta variables are nonpositive, so it's easy to see that the delta substitution Θ that maps every delta variable to 0 satisfies all constraints of the allowed forms. By reasoning similar to that in Theorem 7.2, $\Theta(\widetilde{\mathcal{F}}_0)$ satisfies all the conditions to be a regular generalisation of $\hat{\mathcal{F}}$. But $\Theta(\widetilde{\mathcal{F}}_0) = \hat{\mathcal{F}}$. □

Theorem B.4. *The simplified generalisation system is sound.*

Proof. *Soundness property 1:* Let $\widetilde{\mathcal{F}}'$ be a regular generalisation of $\hat{\mathcal{F}}$. Regularity already requires that all tiles of $\widetilde{\mathcal{F}}'$ have non-negative height and width for every length variable assignment. Since each reference $\widetilde{\rho}$ in $\widetilde{\mathcal{F}}'$ has only one target tile, there is no possibility of overlap between two target tiles of the same reference. It remains to show that for every length variable assignment and with respect to every cell in the specialisation of the calling tile t_c, $\widetilde{\rho}$ evaluates to a range r of non-negative height and width that is covered by the target tile 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 $\widetilde{\rho}$.

- If $\overline{\sigma}$ is of the Inelastic kind, then $\overline{\chi}_1$ and $\overline{\chi}_2$ are stable and t_t is fixed at its height in $\hat{\mathcal{F}}$. Either $\overline{\chi}_1$ and $\overline{\chi}_2$ are both absolute (in which case $\overline{\sigma}$ always evaluates to the same row span it did in $\hat{\mathcal{F}}$) or t_c is fixed at its height in $\hat{\mathcal{F}}$ (in which case, for each caller row, $\overline{\sigma}$ evaluates to the same row span as it did for the same caller row in $\hat{\mathcal{F}}$). Either way, since $\hat{\mathcal{F}}$ is closed, it follows that for every caller row, $\overline{\sigma}$ evaluates 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 variable assignment, t_t and t_c have the same height, and $\overline{\sigma}$ evaluated from 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.
- If $\overline{\sigma}$ is of the Whole kind, then for every length variable assignment, $\overline{\sigma}$ evaluates to the whole row span of t_t, which has non-negative height and is contained in itself.

Soundness properties 2 and 3: Clear for each kind of well-behaved reference.

Soundness property 4: First we prove that if Θ' satisfies the constraints generated for $\widetilde{\rho}$, then $\widetilde{\rho}'$ is well-behaved. Let $\overline{\sigma}' = \overline{\chi}'_1 : \overline{\chi}'_2$ be the row span reference of $\widetilde{\rho}'$; the argument for the column span reference is analogous. Let t_t, t_c, χ_1, χ_2, $\hat{\alpha}_t$, $\hat{\alpha}_c$, $\hat{\alpha}_1$, and $\hat{\alpha}_2$ be defined as in the constraint generator.

- If the algorithm anticipated the Lockstep kind and t_t was ultimately inferred to be vertically inelastic, then we know that t_c and t_t have the same height in $\hat{\mathcal{F}}$, $\chi_1 = \chi_2$, and χ_1 is relative. In order for t_t to be the only target tile of χ_1, χ_1 must have pointed to the top row of t_t in $\hat{\mathcal{F}}$. We generated constraints $\hat{\alpha}_t = \hat{\alpha}_c$ and $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$, so it follows that the heights of t_c and t_t in $\widetilde{\mathcal{F}}'$ are the same non-constant expression, $\overline{\chi}'_1 = \overline{\chi}'_2$, $\overline{\chi}'_1$ is relative, and $\overline{\chi}'_1$ points to the top row of t_t when evaluated at the top row of t_c. Thus $\overline{\sigma}'$ indeed satisfies the conditions of the Lockstep kind.
- If the algorithm anticipated the Lockstep kind but t_t was ultimately inferred to be vertically inelastic, then our constraint $\hat{\alpha}_t = \hat{\alpha}_c$ ensures that t_c is vertically inelastic and this $\overline{\chi}'_1$ and $\overline{\chi}'_2$ are stable. Furthermore, our constraint $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$ ensures that the coordinates of $\overline{\chi}'_1$ and $\overline{\chi}'_2$ are constant. Thus $\overline{\sigma}'$ satisfies the requirements of the Inelastic kind.
- The proof for the remaining cases is similar in spirit.

Now we prove the converse: if $\widetilde{\rho}'$ is well behaved, then Θ' satisfies the constraints generated for $\widetilde{\rho}$. Once again, we consider the row span reference $\overline{\sigma}' = \overline{\chi}'_1 : \overline{\chi}'_2$; the argument for the column span reference is analogous. Let t_t, t_c, χ_1, χ_2, $\hat{\alpha}_t$, $\hat{\alpha}_c$, $\hat{\alpha}_1$, and $\hat{\alpha}_2$ be defined as in the constraint generator.

- If the algorithm anticipated the Lockstep kind, it would constrain $\hat{\alpha}_t = \hat{\alpha}_c$ and $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$. In this case, we know that χ_1 and χ_2 are not superstable, so $\overline{\sigma}'$ cannot be of the Whole kind. If $\overline{\sigma}'$ is of the Lockstep kind, then the heights of t_t and t_c must be the same expression in $\widetilde{\mathcal{F}}'$, so $\hat{\alpha}_t = \hat{\alpha}_c$ is satisfied. Furthermore, $\overline{\chi}'_1$ points to the top row of t_t (which is constant) when evaluated at the top row of t_c for all values of length variables, so $\overline{\chi}'_1$ must be constant and can only be equal to χ_1 since $\widetilde{\mathcal{F}}'$ is a generalisation of $\hat{\mathcal{F}}$, so $\hat{\alpha}_1 = 0$ is satisfied. Finally, $\overline{\chi}'_1 = \overline{\chi}'_2$, so $\hat{\alpha}_1 = \hat{\alpha}_2$ is satisfied. If on the other hand $\overline{\sigma}'$ is of the Inelastic kind, then t_t has constant height in $\widetilde{\mathcal{F}}'$, and $\overline{\chi}'_1$ must be stable despite it being relative, so t_c must have constant height. Furthermore, $\overline{\chi}'_1$ and $\overline{\chi}'_2$ have constant coordinates. Since $\widetilde{\mathcal{F}}'$ is a generalisation of $\hat{\mathcal{F}}$, all these constants must be the same constants as in $\hat{\mathcal{F}}$. Thus Θ' sets $\hat{\alpha}_t = \hat{\alpha}_c = \hat{\alpha}_1 = \hat{\alpha}_2 = 0$, and the constraints $\hat{\alpha}_t = \hat{\alpha}_c$ and $\hat{\alpha}_1 = \hat{\alpha}_2 = 0$ are satisfied.
- The proof for the remaining cases is similar in spirit.

□

Proposition B.6. *In the principal regular generalisation $\widetilde{\mathcal{F}}^*$ of a basic SDF $\hat{\mathcal{F}}$, the height and width of every tile is either a constant or a length variable, not a length variable plus a constant.*

Proof. Let $t_1, \ldots, t_k$ be a maximal set of tiles whose height delta variables $\hat{\alpha}_1, \ldots, \hat{\alpha}_k$ are constrained equal to one another but are not constrained equal to zero. Then we know $t_1, \ldots, t_k$ all have the same initial height (call it h), which is at least 2, so constraint generation step (1) would generate a constraint $h + \hat{\alpha}_i \geq 0$ for every i. Suppose $\hat{\alpha}_1$ is the variable that gets kept by constraint solving step (1). Then constraint solving step (2) will determine that the lower bound of $\hat{\alpha}_1$ is $-h$, and step (3) will replace $\hat{\alpha}_1$ with $\alpha_1 - h$. Thus, the height of each t_i, which was $h + \hat{\alpha}_i$ in $\widetilde{\mathcal{F}_0}$, will simplify to just α_1. As usual, the argument is the same for widths. □

D FULL GENERALISATION SYSTEM

In this section, we describe an extension of the simplified generalisation system to support more interesting SDFs, including those having range references with multiple target tiles. Our motivating example is the COMPOUND SDF of Section 7.4. The elastic SDF that we want is:

```
function COMPOUND⟨α⟩( t_s A3, t_o F3, t_r F1, t_x A4::{α,2} )
  returns F4::{α,1}^t4 {
 t1 C4::{α,1} = A4^tx − A3^ts,tx
 t2 D4::{α,1} = POWER(1+$F$1^tr, C4^t1)
 t3 E4::{α,1} = F3^to,t4 * D4^t2
 t4 F4::{α,1} = E4^t5 + B4^tx
}
```

In general, our main goal is to allow relative references with an upward offset between tiles of similar height, whether or not the tiles start at the same row. So for example, if $\overline{r}_1$ and $\overline{r}_2$ are tiles of size $\{\alpha, 1\}$ (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 reference to the entire target tile and then using the INDEX function to extract the desired element.

The regularity condition 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'd like to add support for cumulative aggregation. For example, we'd like to generalise:

```
function CUMSUM(t1 A1::{3,1}) returns B1::{3,1}^t2 {
 t2 B1::{3,1} = SUM(A$1:A1^t1) }
```

to:

```
function CUMSUM⟨α⟩(t1 A1::{α,1}) returns B1::{α,1}^t2 {
 t2 B1::{α,1} = SUM(A$1:A1^t1) }
```

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 reference in the formula for t_2 has one Start endpoint and one Lockstep endpoint.

In the simplified system, the only kind of row span reference that could contain an unstable row reference was Lockstep, and it required that the heights of the caller 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(t1 A1::{3,1}) returns B3^t3 {
  t2 B1 = A1^t1
  t3 B2::{2,1} = B1^t2,t3 + A2^t1
}
```

(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⟨α⟩(t1 A1::{α+1,1}) returns B{α+1}^t3 {
  t2 B1 = A1^t1
  t3 B2::{α,1} = B1^t2,t3 + A2^t1
}
```

in which the formula for t_3 has an unstable 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.

D.1 Well behaved references

Having decided to allow the caller and target heights of a lockstep reference to differ by a constant, we can proceed to the definition of a well behaved reference. First we need some auxiliary definitions to deal with the multiple target tiles. The *overall target range* of a reference $\hat{\rho}$ in the original SDF $\hat{\mathcal{F}}$ is the range of all cells read by $\hat{\rho}$ with respect to any caller cell. It can be computed as the range from the upper-left corner of $\hat{\rho}$ evaluated at the upper-left corner of the caller to the lower-right corner of $\hat{\rho}$ evaluated at the lower-right corner of the caller. (Since we assume the original SDF is tame, it follows that all of these cells are actually read.) A target tile of $\hat{\rho}$ 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 $\widetilde{\mathcal{F}}$ as a potential generalisation of $\hat{\mathcal{F}}$, a target tile of a reference $\widetilde{\rho}$ is said to be vertically or horizontally final if the corresponding target tile of the corresponding reference in $\hat{\mathcal{F}}$ is. (Trying to determine overlap in the elastic SDF would be more complex because it could depend on the values of length variables.)

Definition D.1 (Well behaved reference in the full system). A range reference $\widetilde{\rho}$ appearing in the formula of a caller tile t_c in an elastic SDF $\widetilde{\mathcal{F}}$ 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. $\widetilde{\rho}$ 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 $\overline{\chi}$ in $\widetilde{\rho}$:
   - *Inelastic*: The height of t_t is constant, $\overline{\chi}$ is stable, and its coordinate is constant.
   - *Lockstep*: The heights of t_c and t_t are non-constant but their difference is a constant, and $\overline{\chi}$ is relative with a constant coordinate.

- *Start*: The height of t_t is non-constant, and $\overline{\chi}$ is superstable and points at a constant non-negative offset above the top row of t_t. Furthermore, if $\overline{\chi}$ is the bottom endpoint of $\widetilde{\rho}$ and the offset is zero, then the height of t_t includes a positive constant.
- *End*: The height of t_t is non-constant, and $\overline{\chi}$ is superstable and points to the bottom row of t_t. Furthermore, if $\overline{\chi}$ is the top endpoint of $\widetilde{\rho}$, then the height of t_t includes a positive constant.

Furthermore, if $\overline{\chi}_1 : \overline{\chi}_2$ is the row span reference of $\widetilde{\rho}$, then the pair of relationships of $\overline{\chi}_1$ and $\overline{\chi}_2$ to t_t must not be (Lockstep, Start) or (End, Lockstep).[16]

D.2 Constraint generation

As in the simplified system, constraint generation is based on the insight that for a given row or column reference $\overline{\chi}$ that is part of a range reference $\widetilde{\rho}$ in $\widetilde{\mathcal{F}_0}$, and a given target tile t_t of $\widetilde{\rho}$, we can narrow the possible relationships of $\overline{\chi}$ to t_t in any regular generalisation of $\hat{\mathcal{F}}$ to Inelastic and at most one other, based on whether $\overline{\chi}$ is relative, the height of the caller tile, and where χ points in relation to t_t in $\hat{\mathcal{F}}$. The constraint generation procedure in detail, for a range reference $\widetilde{\rho}$ in $\widetilde{\mathcal{F}_0}$:

(1) Constrain the height delta variable of each vertically non-final target tile of $\widetilde{\rho}$ equal to 0.
(2) For each vertically final target tile t_t of $\widetilde{\rho}$ and each of the two row references $\overline{\chi}$ that appears in $\widetilde{\rho}$, let $\hat{\alpha}_t$ and $\hat{\alpha}_c$ be the height delta variables of t_t and t_c, let $\hat{\alpha}$ be the delta variable of $\overline{\chi}$, and try the cases below in order. However, if the pair of cases for the two row references in relation to 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 $\overline{\chi}$ is relative and the initial heights of t_t and t_c are at least 2, then we anticipate that the relationship of $\overline{\chi}$ to t_t is Lockstep, but it may turn out to be Inelastic. Constrain $\hat{\alpha}_t = \hat{\alpha}_c$ and $\hat{\alpha} = 0$.
   (b) *(Start)* Otherwise, if the initial height of t_t is at least 2, $\overline{\chi}$ is superstable, and the corresponding χ points at or above the top row of t_t in $\hat{\mathcal{F}}$, then we anticipate the relationship is Start, but it may turn out to be Inelastic. Perform the following steps:
      - Constrain $\hat{\alpha} = 0$.
      - If χ points exactly to the top row of t_t and $\overline{\chi}$ represents the bottom of $\widetilde{\rho}$, then constrain the height of t_t to be at least 1.
   (c) *(End)* Otherwise, if the initial height of t_t is at least 2, $\overline{\chi}$ is superstable, and the corresponding χ points to the bottom row of t_t in $\hat{\mathcal{F}}$, then we anticipate the relationship is End, but it may turn out to be Inelastic. Perform the following steps:
      - Constrain $\hat{\alpha} = \hat{\alpha}_t$.
      - If $\overline{\chi}$ represents the top of $\widetilde{\rho}$, then constrain the height of t_t to be at least 1.
   (d) *(Inelastic)* Otherwise, the relationship must be Inelastic. Constrain $\hat{\alpha}_t = \hat{\alpha} = 0$. In addition, if $\overline{\chi}$ is relative, then constrain $\hat{\alpha}_c = 0$ to ensure that $\overline{\chi}$ will be stable.
(3) Follow the analogues of steps (1) and (2) for the column axis.

D.3 Soundness of the full system

Theorem D.2. *The full generalisation system is sound.*

Proof. Same idea as in the simplified system, just more cases. □

From which it follows that:

[16] If these pairs of relationships occurred, then $\overline{\chi}_1 : \overline{\chi}_2$ would evaluate to a negative height for most rows of t_c when t_c and t_t increase in height. These pairs could otherwise occur in a corner case if t_c and t_t both have initial height exactly 2, while (End, Start) is ruled out by the non-degeneracy condition.

	B	C	D	E
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. 7. A spreadsheet containing the definition of the EXPENSES SDF.

Corollary D.3. *In the full generalisation system, every tame labelled SDF $\hat{\mathcal{F}}$ has a principal regular generalisation $\widetilde{\mathcal{F}}^*$, $\widetilde{\mathcal{F}}^*$ is unambiguous, and the algorithm of Section 7 finds $\widetilde{\mathcal{F}}^*$.*

E ADDITIONAL DESCRIPTION OF USER STUDY

E.1 Participants

We had twenty participants (seven female) aged 18-35 (mean 24), an adequate sample for preliminary field research 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 to high (self-assessed) expertise with spreadsheets, and some to high (self-assessed) programming expertise. Participants were reimbursed with a £40 voucher after completing the study.

E.2 Tasks

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 7 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::{α,1}, t_2 D6::{α,1}) returns E17^{t_4} {
  t_3 E6::{α,1} = IF(D6^{t_2}<C6^{t_1}, D6^{t_2}, C6^{t_1})
  t_4 E17 = SUM(D6::{α,1}^{t_2}) − SUM(E6::{α,1}^{t_3})

```
}
```

This example needs only the simplified generalization system of Section B.
The array SDF is similar and written as:

```
function ARRAY_EXPENSES( C6, D6 ) returns E17 {
 E6 = IF(D6<C6, D6, C6)
 E17 = 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.

(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.

(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, 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.

(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.

(6) Expected profit.
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 8 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 ⟨α⟩ (A2,B1::{1, α},f) returns B2::{1, α} {
 B2::{1, α} = f(A2,B1)
```

	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. 8. A spreadsheet containing the definition of the PROFIT SDF.

}

The elastic SDF is written as:

function PROFIT(t_1 C4::{1,α}, t_2 C6::{1,α} , t_3 B7, t_4 B8)
  returns C12^{t_8} {
 t_5 C7::{1,α} = B7^{t_3,t_5} $*$ (1 + C4^{t_1})
 t_6 C8::{1,α} = B8^{t_4,t_6} $*$ (1 + C4^{t_1})
 t_7 C9::{1,α} = (C6^{t_2} $*$ C7^{t_5}) − C8^{t_6}
 t_8 C12 = SUM(C9::{1,α}^{t_7})
}

This example needs the full generalization system of Appendix D.
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)
}
```

E.3 A note on empirical evaluation

Our study was designed in accordance with the SIGPLAN Empirical Evaluation Guidelines[17]. Our hypotheses and results ('claims') are clearly stated. We chose our experimental alternative (array-based programming) for its external validity; this is the most plausible current alternative, already implemented to varying degrees in commercial spreadsheet packages. The tasks we set our participants (i.e., 'benchmarks') were carefully designed to be representative of real-world tasks, by adapting real spreadsheets. Our sample size was informed by standards for sample size in HCI [Caine 2016], our statistical tests were similarly chosen and we report the distribution of results (Figure 6). TLX and task time are widely-used metrics. Our experimental design is detailed in Section 8.3, and we have taken care to present our results clearly in Section 8.4.

We also adopt practices standard in HCI evaluation not explicitly covered by the guidelines. For example, we counterbalance[18] the tasks, thus mitigating order and learning effects. Although we did not achieve equal gender representation in our sample, women were not significantly

[17] http://sigplan.org/Resources/EmpiricalEvaluation/. Last accessed: March 6, 2019
[18] http://www.unc.edu/courses/2008spring/psyc/270/001/counterbalancing.html. Last accessed: March 6, 2019

under-represented, which was important to account for known gender differences in spreadsheet self-efficacy [Beckwith et al. 2007].

F MOVING TILES TO AVOID OVERLAPS: DETAILS

In this Appendix, we describe the translation of an elastic SDF $\widetilde{\mathcal{F}}$ 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.

F.1 Identifying complexes

Two tiles in $\widetilde{\mathcal{F}}$ are *linked* if they both occur in the target-tile set of some range reference $\widetilde{\rho}$, meaning that we cannot move the tiles to different locations unless we were to replace $\widetilde{\rho}$ 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 Appendix D, tiles t_o and t_4 are linked by the reference $\mathsf{F3}^{t_o,t_4}$ in the tile E4::{α,1}. 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.

F.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 $\widetilde{\mathcal{F}}$ by setting them to their initial values. Specifically, suppose that elastic tiles t_1 and t_2 with ranges $\overline{a}_1 :: \{\overline{l}_{R1}, \overline{l}_{C1}\}$ and $\overline{a}_2 :: \{\overline{l}_{R2}, \overline{l}_{C2}\}$ collide if all length variables are set to ∞. Two rectangles overlap 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 $\hat{\mathcal{F}}$, 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 $\hat{\mathcal{F}}$, then we remove all length variables that appear in $\overline{l}_{R1}$ and $\overline{l}_{R2}$, which will leave the tiles t_1 and t_2 in $\widetilde{\mathcal{F}}$ 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 this never occurs in a principal regular generalisation in the full system.) Conversely, if t_1 and t_2 are disjoint in columns but overlap in rows in $\hat{\mathcal{F}}$, then we remove all length variables that appear in $\overline{l}_{C1}$ and $\overline{l}_{C2}$. If t_1 and t_2 are disjoint in *both* rows and columns in $\hat{\mathcal{F}}$ (e.g., elastic ranges $B4 :: \{1, \alpha\}$ and $D2 :: \{\beta, 1\}$ 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.

F.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 $\overline{x}$ 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 variable 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 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 C's bounding rectangle, and let S be the set of previously processed complexes whose bounding rectangles potentially overlap that of C. If S is empty, then positioning of C is complete, otherwise continue.
(2) Let S_R be the set of complexes in S whose archetypal row spans are disjoint from the archetypal row span of C, and define S_C analogously for columns. Let $S' = S \setminus (S_R \cup S_C)$; a complex may belong to S' if its archetypal bounding rectangle overlaps that of 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 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 C potentially overlaps a given previously processed complex and is moved either down or to the right, 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 F.1. *For every well-defined elastic SDF $\widetilde{\mathcal{F}}$, tile movement produces a well-defined overlap-free extended elastic SDF $\widetilde{\mathcal{F}}_{of}$ that is semantically equivalent to some well-defined elastic SDF $\widetilde{\mathcal{F}}'$ of which $\widetilde{\mathcal{F}}$ is a generalisation.*

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

Contents

Received November 2018