The Lish: A Data Model for Grid Free Spreadsheets

Thesis

School of Computing and Communications
Faculty of Science, Technology, Engineering and Mathematics
The Open University

The Lish:
A Data Model for Grid Free Spreadsheets

Alan Hall

Supervised by:
Michel Wermelinger
Tony Hirst
Santi Phithakkitnukoon

A dissertation submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Submitted September 2019

Abstract

Throughout the history of the spreadsheet, and throughout the majority of research into improving it, the grid of cells has remained a constant as the underlying data model. An idea that has received recent interest is to provide users with a spreadsheet-like environment based on something other than a grid. The attraction is that if salient features of the data structure can be made more explicit, the machine will be able to provide certain types of error checking and automation.

In this project I consider one such grid replacement, a new data model which I call the "lish". It is based on nested lists of cells, composed according to rules that allow repeating structures to be described. It allows columns, tables, groups of tables and other structures to be treated as coherent objects. This supports a novel form of cell range selection, and allows the machine to ensure that related structures are kept consistent. The model is also more accommodating than the grid of dynamic space allocation, where the number of cells occupied by a result is not known in advance.

Then, I develop a "lish calculus", an extension to vector arithmetic for hierarchical structures that provides a concise notation for calculations with lishes. This simplifies the usual spreadsheet formula expressions, and enables the machine to interpret them consistently with the context in which they are located.

I evaluate the lish in the framework of the cognitive dimensions of notations, with the help of example use cases and a user study based on a prototype lish editor. These verify many of the hypothesised advantages, but also reveal some difficulties for users. I close with an analysis of how the lish might be revised to address these shortcomings, while continuing to capitalise on the essential benefits.

Acknowledgments

First, I would like to thank my three supervisors, Michel, Tony and Santi for all their patience, advice and guidance through the course of this research.

I gratefully acknowledge the financial support provided by my employer, the Government Operational Research Service (GORS). I would also like to thank GORS colleagues, both for their support and understanding through the challenges of doing research in parallel with "the day job", and for their thoughtful engagement with the user study.

Contents

Previously published work

This dissertation includes work that has previously appeared in the following publications:

> Hall, A. G., Wermelinger, M., Hirst, T. and Phithakkitnukoon, S. Structuring spreadsheets with the "lish" data model. In *Proceedings of the European Spreadsheet Risks Interests Group (EuSpRIG)*, 2017.
>
> Hall, A. G., Wermelinger, M., Hirst, T. and Phithakkitnukoon, S. Wide, long, or nested data? Reconciling the machine and human viewpoints. In *Proceedings of the Psychology of Programming Interest Group (PPIG)*, 2018.

Chapter 1

Introduction

1.1 The spreadsheet data model: grids of cells

Data in tabular form are everywhere: official statistics, company accounts, scientific results – to name but three examples. For statistical analysts and data scientists, we might say that the table is the raw material of their labours. Despite the fast-growing uptake of programming languages such as R for this purpose [Muenchen, 2015] one of the most widely used tools has long been, and remains, the spreadsheet [Scaffidi, 2016].

Although the directness and usability of the spreadsheet make it appealing to users [*ibid*], it has gained a certain notoriety for being error-prone [EuSpRIG, 2019]. Previous research has investigated a range of approaches for reducing the risks of errors. These have included generating the sheet from a template to ensure consistency [Engels and Erwig, 2005], creating visualisations of the data flow [Hermans et al., 2011], and closer integration with external databases [Bakke and Karger, 2016].

One aspect that the vast majority of this prior work has in common is that it retains the rectangular grid of cells as the underlying data model of the spreadsheet. An early exception was the work of Burnett et al. [2001], who used a system of "forms" and "dynamic grids". More recently this aspect has received renewed interest, associated with bringing spreadsheet-like approaches to database applications (for example, McCutchen et al. [2016]).

Perhaps it is not altogether surprising that the underlying grid of cells has not been seen as an obvious candidate for improvement. On the face of it, the grid is really rather a good representation of a table, so why would we want to change it? Even if there is a non-grid arrangement of the cells

Title: Quarterly A&E Activity and Emergency Admissions statistics, NHS and independent sector organisations in England

Summary: A&E performance and emergency activity quarterly time series

Period: April 2004 - Quarterly to date
Source: Unify2 data collection - WSitAE, QMAE
Basis: Provider
Published: 7th April 2015
Revised:
Status: Published
Contact: Paul Steele - Unify2@dh.gsi.gov.uk

England Level Data

			A&E attendances				A&E attendances > 4 hours from arrival to admission, transfer or discharge					
Area Team	Year	Quarter	Type 1 Departments - Major A&E	Type 2 Departments - Single Specialty	Type 3 Departments - Other A&E/Minor Injury Unit	Total attendances	Type 1 Departments - Major A&E	Type 2 Departments - Single Specialty	Type 3 Departments - Other A&E/Minor Injury Unit	Total Attendances > 4 hours	Percentage in 4 hours or less (type 1)	Percentage in 4 hours or less (all)
-	2011-12	Q1: April - June	3,586,381	165,665	1,749,691	5,501,737	159,256	868	2,224	162,348	95.6%	97.0%
-	-	Q2: July - Sept	3,469,728	158,396	1,716,708	5,344,832	142,246	499	2,098	144,843	95.9%	97.3%
-	-	Q3: Oct - Dec	3,432,578	152,074	1,668,923	5,253,575	189,038	451	2,431	191,920	94.5%	96.3%
-	-	Q4: Jan - Mar	3,525,235	161,861	1,694,162	5,381,258	222,749	424	2,659	225,832	93.7%	95.8%
-	2012-13	Q1: April - June	3,623,761	162,078	1,801,868	5,587,707	184,483	511	2,758	187,752	94.9%	96.6%
-	-	Q2: July - Sept	3,582,519	157,959	1,753,166	5,493,644	165,139	444	2,338	167,921	95.4%	96.9%
-	-	Q3: Oct - Dec	3,545,721	151,314	1,654,412	5,351,447	228,920	545	2,504	231,969	93.5%	95.7%
-	-	Q4: Jan - Mar	3,500,067	154,175	1,651,597	5,305,839	310,035	729	3,005	313,769	91.1%	94.1%
-	2013-14	Q1: April - June	3,620,189	163,624	1,770,314	5,554,127	237,553	816	2,916	241,285	93.4%	95.7%
-	-	Q2: July - Sep	3,588,450	159,339	1,777,362	5,525,151	202,551	600	2,648	205,799	94.4%	96.3%
-	-	Q3: Oct - Dec	3,476,480	150,033	1,667,756	5,294,269	227,400	336	2,756	230,492	93.5%	95.6%
-	-	Q4: Jan - Mar	3,528,029	155,608	1,721,473	5,405,110	257,815	551	3,244	261,610	92.7%	95.2%

Figure 1.1: Extract from an example spreadsheet: NHS statistics on Accident & Emergency admissions. Source: NHS England. Contains public sector information licensed under the Open Government Licence v3.0.

that somehow better follows the contours of the data, it is not obvious how this would actually reduce errors or otherwise benefit the user.

If spreadsheets consisted only of relational tables, then there would indeed be little need to look beyond the existing grid. But few spreadsheets are quite as simple as that. Consider the example in Figure 1.1, which shows some official statistics published by NHS England. In addition to the main table, this spreadsheet contains various metadata which are colocated in the same document. Not only that, but the main table contains considerable additional structure over and above its obvious arrangement in rows and columns. The columns have been formed into groups (with a merged cell spanning each group at the top); likewise there is an implicit grouping in the rows, of quarters within years. And there are repeating patterns where certain regions of the sheet express the same calculated relationship with their neighbours.

The accident and emergency data example is a fairly simple spreadsheet, with only one table. More complicated spreadsheet models might have many tables, among which some might be required to have the same structure as each other, and to carry equivalent calculations. The simple spreadsheet grid can of course accommodate such arrangements – indeed, that is one of its strengths. But of itself the grid does not "know" about any of the repeating patterns or their associated constraints.

1.2 The "lish" data model: lists of cells

In this project I investigate an alternative data model, called the "lish", for spreadsheet-like data. Instead of a grid of cells, the lish is based upon lists of cells; these lists may be nested, to arbitrary depth. The lish is aimed at expert analysts rather than casual users – first, because they are likely to be dealing with more complicated models, and second, because this user group might be expected to take a more systematic approach towards model structure.

It is worth noting that the conventional spreadsheet grid is a special case of the lish: it can be represented as a list of lists of cells, in which each of the inner lists has the same length. The lish, however, is a more general model, which can capture more of the internal structures and patterns that I have been describing.

But even if a lish representation is a better match for the structure of the data, is this really any more than an aesthetic issue? I shall argue that it is. My rationale is that if the machine has more information about the intended structure, it can be made to apply this information in ways that serve the user. I will elaborate upon this in a more extended example, which I defer until section 3.1. Briefly, though, the main advantages that I will be claiming for the lish are:

- requiring both fewer and simpler formulae;
- maintaining consistency in related parts of the structure; and
- facilitating interactive cell range selections.

Simply transferring a normal spreadsheet into a list of cells is not the whole story, however. First, in moving to a list we appear to have discarded important geometrical information: how do we ensure that lists that are meant to represent tables appear as such? And conversely, for lists where there is not a tabular interpretation, how do we suppress spurious alignment? Second, a list representation on its own fails to capture some of the basic constraints that we need when working with tables. For example, if we construct a table as a list of lists, where each inner list corresponds to one row, then we need to ensure that all these inner lists are the same length. This leads to my first research question:

RQ1. How can we model spreadsheet-like data using a representation based on lists of cells?

In Chapter 4, I will develop the theory for the lish representation, taking into account the constraints noted above, and in Chapter 5 I will address the question of how to visualise a lish. Then in Chapter 6, I will extend the theory to support spreadsheet-like formulae, by defining a lish calculus. This builds on the idea of vectorised calculation used in the R programming language [R Core Team, 2018]. In R, vectorised calculations are defined on one-dimensional lists (vectors) and regular arrays; lish calculus extends their principles to arbitrarily nested structures.

As part of this project I have developed a prototype lish editor. This provides a somewhat spreadsheet-like environment in which a user may enter data in lish form. The editor also implements lish calculus, so the user may enter cell formulae in order to carry out calculations.

I evaluate the lish representation using the cognitive dimensions framework of Green and Petre [1996]. To this end I have created some example applications of the lish which I report in Chapter 7, and conducted a user study which I report in Chapter 8. These parts of the work aim to answer two further research questions, namely:

RQ2. What would be the consequences for analytical workflow of using a lish, instead of a grid, as the basis of a spreadsheet-like environment?

RQ3. Where would this alternative representation be located in the space of the cognitive dimensions?

Chapter 2

Tables and spreadsheets: a literature review

2.1 Organising data in tables

Humans have been recording data in tables for at least three thousand years; Friberg [1981] describes an example on a clay tablet from ancient Babylon which has been dated prior to 1600BCE. The Oxford English Dictionary defines a table as:

> "A set of facts or figures systematically displayed, especially in columns."

If we want to apply this definition in computing, much hangs upon the word "systematically"! But although this is a broadly drawn, everyday language definition, it actually sits rather well with the spirit of the spreadsheet, where the "system" of organisation is at the discretion of the user. I shall return to the spreadsheet itself in section 2.3; but first, I shall look a little more closely at the meaning of a "table" in computer science.

2.1.1 Representing tables in a computer

How can we represent a table in computer code? The basic building block is the *array*, which has been available since the earliest programming languages, for example in FORTRAN [Backus et al., 1957] and Algol 60 [Backus et al., 1963]. Modern dynamically typed languages such as JavaScript, Python and Ruby provide lists (numerically indexed arrays) and hash tables (associative arrays) as their basic container types. More complex structures can be constructed from these building blocks. The JSON file format

[ECMA, 2013] is closely related to the JavaScript representation of these object types; A W3C recommendation [Tandy and Herman, 2015] is based on JSON but extends the notation to include metadata. YAML [Ben-Kiki and Evans, 2009] uses a very similar representation but with a more human-readable syntax.

Lists and associative arrays are not tables in the general sense, but allow tables to be represented as a list of records. Each record represents one row, and is in the form of an associative array with one element per field. A table so constructed does not natively support operations such as inserting a whole column, but libraries are available to provide such functionality. The NumPy package for Python [Oliphant et al., 2014] is one such library. It uses the array of records scheme, with an auxiliary "dtype" object to hold the column specifications. A "column" within the dtype object may itself be an array, providing support for nesting columns, such as quarters within years. The Pandas toolkit [McKinney and PyData Development Team, 2014] provides additional data analysis functions and is built on top of NumPy. In terms of data representation, the notable addition is a panel data type for 3D data, formed out of an array of tables. It is also possible to represent higher dimensional data, but the user must first use a class factory to create a class with the appropriate number of dimensions.

A common scenario with statistical data is that different columns may be of different data types (integer, floating point, string, etc.) but the data type is consistent within each column. An example was seen in Figure 1.1 with the NHS statistics. This provides flexibility, while allowing memory allocation to be optimised. The R language [R Core Team, 2018] was designed expressly for statistical analysis, and supports tables of this kind natively in the form of the "data frame". Unusually, the underlying storage of this class uses column-major order – that is, all the values in the first column, followed by all those in the second, and so on – as opposed to row by row. It is represented as a list of vectors (where a vector in R contains elements all of the same type, whereas a list may freely mix types).

There are fewer available representations for data that span more than two dimensions. One option is to use more deeply nested lists in JSON or similar, but the associated code needs to process the nested levels in a way that respects their intended interpretation. R and many other languages have native support for arrays with any number of dimensions, but every element of the array must be of the same type. Pandas also has some

support for higher dimensional data as noted above.

For data that are not purely rectangular ("ragged" arrays), the dynamic languages mentioned provide a high degree of flexibility for building custom classes. For example, suppose we wanted to represent in a single object *all* of the Accident & Emergency data from the introduction, including the title, summary, period, and other metadata. We could define a Python class having a Pandas table property to represent the main table, and string properties to represent the other attributes. The ability to capture fully the structure of custom objects in this way is one of the strengths of scripting languages, but comes at the expense of directness: unless one develops an editor for them, the objects belonging to these custom classes cannot be edited directly by the user.

Looking ahead to the spreadsheet, I note that it allows three dimensional data to be represented as a "family" of two dimensional tables, which can be visualised as a stack of planes. The spreadsheet does not "know" however that these tables are part of the same object, and will not ensure that they are treated in a consistent way. An early proposal from Furuta [1986] was to capture higher dimensions using a tree-like structure with tables as the leaves, but this does not appear to accommodate the scenario where some of those leaves are planes within a 3D array.

The lish as I have introduced it briefly in the introduction is based on a different organising principle: it arranges cells in nested lists. These cell groupings allow one to identify patterns where parts of the data have common properties. It is this feature that allows tables to be represented, but the lish does not impose any definition of what a table is. It is a model *for* tabular data (and other forms besides), rather than a model *of* tabular data.

2.1.2 What can we learn from typesetting?

In this project I am concerned with structuring tabular data for use in processing. The concerns of typesetting are more aesthetic, so I shall confine my attention to a very narrow question: can the conventions of typesetting tell us anything about the structure that is being assumed in the table being set?

Let us begin (of course) with LaTeX. Tables here are based on a grid, rather as in a spreadsheet. The equivalent of merged cells is also possible, to obtain headings that span multiple rows, or columns. The user specifies

the contents of the table cell by cell. Column widths are determined automatically. Many additional packages are available for tuning various aspects of alignment and spacing, and for setting multi page tables. A notable and comprehensive example is the Booktabs package [Fear, 2016], which produces an output closer to the historical conventions of typesetting. It would be fair to say that creating tables requires the user to work at a lower level of abstraction than with other common document elements. For example, a column heading is not a separate abstraction, but just the first row on the grid. Any visual differentiation of this row must be specified explicitly. Similarly if there is structure such as grouping within the table, the user may insert gridlines or extra spacing to clarify this structure, but there is no abstract concept of a "group" of rows or columns.

By contrast, Wang [1996] proposes an approach to high quality typesetting that starts with a more abstract model of a table. In Wang's model, topology is separate from the abstract model and style is separate again, so the same abstract table could be presented visually in numerous ways. Wang defines categories (from which the table margins are to be composed) in terms of sets and sequences; the categories can be nested, for example to form column groups. Body elements are then defined in terms of their coordinates along the categories. The *stub* and the *boxhead* (terms based on the Chicago Manual of Style) hold row and column headings respectively. This model comprising stub, boxhead and table body appears well-suited to standalone tables in print media but may be insufficiently general for the spreadsheet, where boxhead type material may be embedded within the sheet.

Anglim [2009] proposes a "Grammar of Tables" which similarly separates table structure from formatting.[1] The level of abstraction is higher than in LaTeX, but the specification of layout appears to be more explicit than in the Wang model. A similar approach is taken by Hlavac [2016] in Tablemaker, an Excel extension for production quality tables. Once again, it facilitates the user treating their table at an abstract level in terms of its headings, body and page layout; appropriate formatting is generated automatically based on the user's choices.

The lish, of course, is intended for arranging data for calculation, rather than typesetting. It shares some similarity with the approach of Wang [1996] reviewed above, in that it defines an abstract model of the data that is

[1] An R package of the same name is unconnected; the "grammar" referred to in this package is a syntax for specifying statistical summaries to be generated.

independent of any geometry. And we will see in section 3.1 that "template cells" in a lish behave somewhat like the boxhead in a classical typeset table. These cells however can appear anywhere within the structure, not just in the row and column margins; it is this feature that allows the lish to represent collections of separate tables (as might appear juxtaposed on a spreadsheet grid). For the user to interact with a lish in a spreadsheet-like way, a concrete visualisation is still necessary. How to provide it will be the subject of Chapter 5.

2.1.3 The semantic view of a table

Up to this point, I have treated the tabular grid as a kind of substrate, where items may be arranged in rows and columns, but any meaning attached to that arrangement is entirely user-defined. This is more or less the spreadsheet position. An alternative treatment is to place a specific semantic interpretation on the table. The more abstract typesetting models above have already taken us a step in this direction, since there is an implication that if a table body cell A lies at the intersection of a boxhead cell B and a stub cell C, then the value of A represents some *fact* about B and C.

This kind of view of a table was originally crystallised in the relational model [Codd, 1970]. A relational table contains information about a single type of entity. Each row in the table describes one entity of that type; columns may be key columns, which collectively identify the entity in question, or attribute columns, which record information about that entity. The theory extends to how entities may be related to other entities in the same or a different table, by defining a *foreign key* to link the two.

A modern view of the relational model through the lens of data science has been taken by Wickham [2007, 2014]. Wickham's focus is on *tidy* data. Tidiness is similar to normalisation in the language of the relational model. Tidy data provide a solid basis for a statistical toolchain, since if analysis procedures both consume and output their data in tidy form, unnecessary and tedious extra processing in between analytical steps can be eliminated.

One of the issues that Wickham addresses is that in many statistical datasets, especially those involving time series, there is a tension between representing data in *wide* or *long* form (I will later come to an example of each, in Figures 3.1 and 3.3 respectively). The former is generally more human-readable, but the latter is "tidier" and often more suitable for machine processing, a distinction I test directly as part of my user study in

section 8.3.1.

An alternative treatment of the wide versus long question is offered by Mount and Zumel [2017]. They define a more abstract description of a table in the form of *coordinatized data*, where the position of an item in a table is defined in terms of its levels along various dimensions. The dimensions correspond broadly to the fields of a composite key in the relational model, and hence identify an entity independently of the physical layout of the table. They demonstrate an example where the same data can be arranged in several different layouts, but the analysis to be carried out has a common expression in coordinatized notation for all of those layouts.

A similar idea has been applied to JSON data in the form of JSON-stat [Badosa, 2015], which is used by the UK Office for National Statistics, among others, as a data interchange format. JSON-stat defines a cube model which maps a flat array for the values in the table body to dimensions and levels. This is not only more compact, but expresses the dimensional structure more explicitly than an array of arrays would do.

The lish does not impose any semantic interpretation on the data. The closest approach it gets is in an implicit (though not mandatory) interpretation of template cells as labelling some set of other cells, which can then be treated as a single object. Whether this object is an entity, an attribute, or neither is for the user to decide freely as they design their model. The lish has no native concept of a foreign key relationship (I consider briefly an extension in this direction in subsection 6.9.2). So comparing it to a relational database, the lish has a different representation for the tabular part, but does not yet seek to address the the inter-table relationship part.

In common with the model of Mount and Zumel [2017] above, the lish allows data to be processed in either long or wide format without the need for an explicit transformation between the two. But in the lish, the dimensions are implicit in the level of nesting of the structure and not a separate abstraction.

2.2 End user development

End user development (EUD) is a large topic, of which spreadsheet use forms only a part. The latter is my main focus in this project, so in this section I shall review the wider area only selectively. I shall begin with some works that investigate the particular requirements of end users. I shall then consider some of the alternatives to spreadsheets for analytical use, and

whether these can tell us anything about how the spreadsheet itself might be improved.

2.2.1 The requirements of analytical EUD

Lieberman et al. [2006] define end user development as:

> "A set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artefact."

Ko et al. [2011] give a somewhat narrower definition, but one which aligns more closely with EUD as practised by analysts, namely:

> "Programming to achieve the result of a program primarily for personal, rather than public use."

For analysts, though, this latter definition must be modified in one regard: it is the program itself that is only for the analyst's use – the results could well be for public use, or at least for use by a third party client. Ko et al. go on to review the characteristics typically found in systems aimed at an EUD audience: visualisation (especially in the context of dependency analysis); provisionality, where parts of a design may be imprecisely stated; programming by example; and the orientation of debugging tools towards "why" questions as opposed to "what-if". They point out the tension between formality and accessibility, and hence the benefits in resolving it of notations that are both accessible and precise. Similarly, Ryder et al. [2005] highlight the four features of concreteness, directness, explicitness and immediate visual feedback.

Holwerda [2017] investigates what professional end user developers might gain from block-based languages used to teach children programming. He found that certain aspects carried over well. Direct structural manipulation was useful for this group. Another advantage was that the graphical UI made it easy for the user to discover the available components: "users can rely on recognition instead of active recall". But Holwerda sounds a note of caution that seemingly friendlier approaches like drag-and-drop can actually be a barrier, as they become cumbersome if the user already knows the command they need. Interestingly, users found it easier to scan the accompanying code than a block-based diagram.

Much of what can be said for a general EUD audience goes for EUD analysts as well. Kery [2017] observed in studies with analysts that programming for them is an exploratory process, which tends to generate many minor variants of the same code. Examples might be data cleaning and model fitting, where decisions on how to proceed at each stage are dependent on what was discovered about the data at the previous stage. This suggests that, among the attributes noted above, provisionality and immediate visual feedback are particularly important.

If the analyst later decides to proceed to large scale automation, then programming by example becomes relevant as well, with the steps used during the exploratory phase on specific data forming the template for future cases. Lee et al. [2017] notes the "flash fill" facility in Excel (where the machine infers from the start of a sequence how to continue it) as a simple instance of programming by example, but observes that in more complicated applications a frequent problem is ambiguity in the user-provided examples from which the machine is attempting to generalise.

Several criteria for evaluating a statistical computing tool are provided by McNamara [2018]. These include:

- The ability to treat results as data. This allows analyses to be chained together; certain older tools produced their results as textual reports, making such chaining difficult to automate.
- Support for an exploratory cycle of analysis.
- Support for associated narrative, and reproducibility. The "Notebook" type of environment reviewed below is aimed at fulfilling this requirement.

McNamara also suggests that users may prefer hierarchical visualisations to "tidy" ones.

2.2.2 Coding for analysts

End user development need not exclude the writing of actual code. The distinction in this context is not between novice and expert programmers, but between programs written as a product and programs written to find the answer to an analytical question. The analyst may well accept the extra cognitive load of traditional coding in order to gain advantages such as more powerful automation and a more rigorous audit trail.

Muenchen [2015] has surveyed the use of software by data analysts across a number of dimensions, including job adverts, scholarly articles, forum activity and user surveys. He found significant use of general purpose programming languages such as Java, C and its derivatives, but across most dimensions R, SAS, and SPSS were dominant. In certain dimensions a number of other languages were found to be of importance, including MATLAB, Python and Stata.

General purpose programming languages, then, remain one of the options available to analysts, but domain specific languages are more typically used; these languages are more oriented to the needs of what Chambers [1998] describes as "programming with data". Similarly the analyst does not strictly require a specialised integrated development environment, but some of these are now available – for example, Rodeo (for Python) and RStudio (for R). These IDEs add integrated plotting and data browsing to the usual coding facilities.

Given the exploratory nature of a lot of analytical work, "live programming" might be expected to be of interest to analysts. Tanimoto [2013] characterises live programming by the ability to edit the program even while it is executing, as opposed to requiring distinct phases for edit, compile, link and run. I interpret the term here to include any form of programming in which the results are immediately visible. In its simplest form, this could be just the REPL (read, execute, print, loop) of the traditional console. Arguably that is not really "programming" as it is limited to one line at a time, but the "read" could refer to a small multi-line fragment that is then executed. Similarly, an interpreted script could be regarded as somewhat live in comparison to a compiled program, but there is still a gap between the programmer composing a sequence of operations and seeing their result.

This idea of decomposing a program into manageable chunks, whose intermediate outputs can be viewed at each stage, is the basis of the Jupyter Notebook. Additionally, this system allows rich text and graphics to be interleaved with the code. Guo [2013] has commented on how this arrangement is particularly well suited to the analytical workflow. The integration between code and data can be tighter (in the sense of more visible to the user) with this system than with a general purpose programming language, but the data still reside in a workspace which remains invisible until the user issues explicit commands to inspect it. The Stencilla document editing system [Aufreiter et al., 2018] performs a similar role, while aiming to feel

more like an office suite.

A further level of liveness may be introduced by allowing the user to see the results of code even while it is being edited. Hundhausen and Brown [2007] present evidence to support the effectiveness of edit-time feedback. They address the problem of implementing liveness in a function definition (whose arguments will not be known until run time) by assigning default values as a starting point. This chimes with the proposal of Victor [2012] that users should be allowed to "start constant, then vary". Victor also observes that (inexperienced) users typically create by reacting to the results of what they see, more than by designing in advance, and advocates eliminating hidden state wherever possible. Live programming might be expected to help with both of these aims.

Granger [2012] presents the Light Table IDE, which uses a drafting table analogue to present code and data to the programmer. Instead of arranging the materials by file, the editor inspects the piece of code the programmer is currently working on and automatically arranges around it other code and documentation relating to the functions it calls, as well as a live view of the results of the code. Code Canvas [DeLine and Rowan, 2010] takes a similar approach, but based on a zoomable view of the entire project, with careful control of the level of detail displayed as the zoom is adjusted. Both of these highlight the importance of using the spatial layout intelligently to juxtapose relevant information.

Another interesting proposal comes from French [2013] in the form of the Larch editor for partially visual programming. This has similarities to the notebook approach described above but makes greater use of spatial layout: its layout engine can arrange and wrap material in the available display space rather than following a simple linear flow as in the notebook. In its present form it appears targeted more at rapid GUI development than at analysis – its aim is "code as literature" (in the author's words), rather than data as literature.

Another angle on the compromises between scripting and direct manipulation is offered by Chugh et al. [2016], this time in the context of a drawing program. The user can both draw interactively (the actions being recorded as macros) and write scripts. Their innovation is to have the machine work out intelligently how to modify an existing script in response to an interactive modification to the drawing, as opposed to retaining the original operation and representing its amendment naïvely as an incremental

operation.

2.2.3 GUIs for analysts

Several commercial statistics packages (for example SPSS, Minitab and Statistica) use a well-established graphical user interface model in which the data are presented in a grid, where rows represent cases and columns represent variables, and operations are invoked from a menu or ribbon bar. This model is a good fit for the exploratory type of workflow described above; scripts are available as alternatives where automation is required. Valero-Mora and Ledesma [2012] survey the GUIs for R, which was originally command line only. They make the point that the command line can be more productive if the initial learning curve is excluded. Similarly Unwin [2012] discusses the pros and cons of the GUI versus command line for statistical work, arguing that the former may be preferable when "flexibility and immediacy are more important than tight control and precision". The command line forces the user to adopt a more formal mental view of the data structure, but once this discipline has been achieved, more powerful operations become possible.

One of the ways in which a graphical interface can be easier on the user than code is in its greater use of spatial information. Kandogan et al. [2011] studied the importance of spatial arrangement to humans using computers as problem-solving aids while Kirsh [1995] presents a more general theory of the different ways in which humans can use spatial layout as a cognitive aid.

2.2.4 Lessons for the lish

The lish shares with the spreadsheet many of the attributes desirable for end user development reviewed in this section. Since it shares a spreadsheet-like interaction model, it benefits from the properties of directness and immediate visual feedback. Also like the spreadsheet, it is well suited to models of a provisional or exploratory nature, and benefits from the use of spatial layout to help the user visualise the structure of their model.

The lish additionally captures a few other benefits pointed to by the EUD literature. As a structured representation, it aims to find the sweet spot described by Ko et al. [2011] for a notation that is both accessible and precise. In its support for hierarchical data, it accords with the suggestion of McNamara [2018] that users should be able to view data in this form.

It also has its own version of "liveness". In common with the spreadsheet, formula evaluation is live in the sense that when a value is altered by the user, all downstream calculations are immediately updated. The lish takes this further by allowing the structure to be the result of a formula as well, so that too is live.

2.3 Spreadsheets

2.3.1 Introduction

From the first release of VisiCalc in 1979 [Bricklin, 2009], through the Lotus 1-2-3 era of the 1980s and into the dominance of Excel to the present day, the spreadsheet has been one of the most successful software applications in history [Scaffidi, 2016]. Scaffidi highlights the spreadsheet's sheer *usability* – especially, arising from its use of direct manipulation – as a reason for its popularity, and points to this rather than perfect optimisation as a major driver of its success.

Although modern spreadsheets provide an ever-increasing array of features, they remain based on two simple and easily comprehensible abstractions: the cell, which may typically contain a single numeric, string or boolean value; and the grid, which arranges the cells in a rectangular array. The value of each cell may be either a literal, or the result of evaluating a formula, which usually refers to other cells. The user can take advantage of the spatial arrangement to visualise the relationships between different items of data.

This basic underlying technology of a grid of cells has endured throughout. Sestoft [2012] has published a detailed technical paper on the considerable challenges in implementing this deceptively simple system efficiently. A review of these issues including some modern approaches to extending the technology is given by Bock [2016].

2.3.2 Spreadsheet errors

The rise of the spreadsheet has however been tarnished by its reputation for being error-prone, with sometimes calamitous results [Panko, 2015; EuSpRIG, 2019]. It is perhaps unfair to level this charge solely at the spreadsheet itself, because one cause of the errors is the context in which it is used. The spreadsheet is so accessible that anybody can use it, and anybody does! As Panko points out, the practices that exist for quality assurance in profes-

sional coding are often lacking among spreadsheet users, making it easier for errors to slip through regardless of the safety or otherwise of the spreadsheet itself. Nevertheless, it is not difficult to find ways in which the spreadsheet is intrinsically error-prone, and much of the research into spreadsheets has been concerned with mitigating this problem.

Panko and Aurigemma [2010] distinguish between qualitative (latent) errors and quantitative errors, giving a detailed taxonomy of the latter, which they divide into *planning errors* and *execution errors*.

Planning errors are divided further into *domain planning* (from knowledge of the application area being modelled) and *spreadsheet planning* (from knowledge of spreadsheet technology, e.g. use of formulae). Domain planning errors appear to be beyond the reach of any analytical tool to remedy, except maybe insofar as a cumbersome tool might divert mental resources that could otherwise be directed to their prevention. Spreadsheet planning errors appear a more promising ground for research, and indeed one of the aims of the lish is to simplify formulae. Panko and Aurigemma specifically mention "non-2D logic" in this category; we shall see in section 7.4 an example where a lish formula is scaled gracefully from two to three dimensions, helping to address this cause of error.

Execution errors are once again divided into two categories. The first is the *slip*, defined as a sensory-motor error, such as a pointing error. The range selection mechanism provided by the lish (subsection 5.4.3) might be of some assistance here, by preventing an "out-by-one" error arising from selecting not quite all the cells in an intended range. The other category is the *lapse*, defined as a failure in (human) memory. Although this category appears harder to address directly in tool design, approaches such as ensuring that related information can be seen juxtaposed might mitigate it by avoiding the need for the user to hold so much in memory.

The qualitative side has been taken up by Leon et al. [2015]. Qualitative errors are risky practices that do not immediately result in an incorrect calculation, but make the spreadsheet hard to understand or brittle to future development. Examples are poor labelling and hard-coding of constants in formulae. Leon defines four top level categories:

- formula integrity
- semantics
- extendibility

- logic transparency

Two of these categories I specifically address with the lish. Under *formula integrity*, Leon notes the frequent use of formula replication in spreadsheets. There is a risk that a failure to replicate over the entirety of the intended range, or an accidental edit to a formula so that it is no longer a faithful replicate, could give rise to error. The use of vectorisation in the lish (to be introduced in section 6.4), so that one formula does the duty of numerous replicates in a normal spreadsheet, tackles this problem.

The lish also addresses the *extendibility* category. Two of the problems Leon mentions here are incorrect use of relative versus absolute references in formulae, and a lack of robustness to adding an extra dimension to a model. In preventing the first of these, vectorisation is again a powerful weapon, especially when combined with the structural deductions that will be detailed in subsections 6.7.4 and 6.8.4. The second one arises fundamentally from the two dimensional nature of the spreadsheet grid. In the lish, an extra dimension is simply an extra level of nesting, so scaling a model from two to three dimensions or higher is comparatively straightforward.

Related to qualitative errors is the notion of "code smells" as they apply to spreadsheets. Hermans et al. [2012] propose several criteria which might indicate that a cell formula is "high risk". The results are displayed as a heatmap, allowing the most problematic areas of the sheet to be easily visualised. Abreu et al. [2014] combine this approach with fault localisation, and have produced a catalogue of "smells" based on the literature.

In the remainder of this chapter, I shall be surveying numerous enhancements that have been proposed or implemented in the spreadsheet to reduce the risk of these errors, or otherwise improve the capabilities of the tool. But first, we might ask whether spreadsheet errors can be reduced by instilling good practice in users, without requiring new technology as such.

An early example of good practice guidance was published by Conway and Ragsdale [1997], who contrast the readability of different spreadsheet layouts for the same model. This appears to be the earliest reference to the "left to right, top to bottom" rule of thumb for spreadsheet design – they observe that users tend to scan in that direction, so the order of processing should flow accordingly. They advocate preferring the form that best *communicates* the spreadsheet's purpose over adhering rigidly to a "standard form". Raffensperger [2001] agrees, and makes a forthright case against a "computer programming" style of spreadsheet writing: "...spreadsheet

users do not use and do not want this style, and they do not want imposed standards." Existing guidances forced users to think in terms of inputs, calculation and outputs, and not in terms of the business model. Raffensperger advocates instead treating a spreadsheet as a piece of mathematical writing, and thinking in terms of authors and readers, not developers and users. This perspective of seeing analysis as literature has parallels with the Notebook style of working for analytical code described earlier.

Grossman and Ozluk [2001] compare three published standards for good practices in professional spreadsheet production, published respectively by FAST, Operis and SSRB. Common themes include a standardised approach to sheet layout, a modular design with designated sheets for designated purposes, and various aspects of labelling and formula hygiene. These standards deal primarily with the mechanics of spreadsheet construction; Leon et al. [2010] reveiew the role of governance and quality assurance in ensuring correct spreadsheets.

The Sprego methodology [Csernoch and Biró, 2015] similarly seeks to instil good practice (in students) with a standardised approach, focusing on incremental problem-solving and simplicity. It concentrates on a manageable subset of the available worksheet functions and makes extensive use of array formulae.

2.3.3 Inferential tools

The spreadsheet as seen by the machine is just a collection of cells, but the user probably has higher abstractions in mind. The cells might for example represent a list, a time series, a table or a matrix. If the machine "knows" what these abstractions are, this information might be used to support the user in various ways. Collections of cells that represent a single object can be treated as such, preserving that object's integrity. Certain classes of errors can be detected: for example, a column in a table where one cell contains a formula that is inconsistent with the others.

The support that the lish can provide to the user is enabled in part by the way in which it makes some of these cell groupings explicit. But even in a normal spreadsheet, where no such explicit grouping is present, one approach that has been extensively utilised is to have the machine *infer* it. There are some conventions as to how data are laid out on a spreadsheet which are not mandatory but are commonly observed. For example, a table often consists of one row of textual labels, followed by columns containing

the actual data.

One of the first studies to formalise this approach and apply it at scale was by Mittermeir and Clermont [2002], who carried out an industrial case study applying a tool they developed to 78 spreadsheets captured in the wild. They postulate a conceptual model on the part of the spreadsheet developer that is not naïve (these are professionals in their own domain, just not professional software engineers), but may not be what the software engineer would expect. Their approach is to try to elucidate this model from the spreadsheet as created. They introduce *logical areas* and *semantic classes* as abstractions over the spreadsheet. These abstractions identify regions where the formulae appear to have been copied from a common origin, or more generally where there is some form of structural similarity – as the authors put it, "hierarchies of increasingly larger, consistently replicated portions of a sheet can be identified." We shall see that it is just such hierarchies that the lish aims to capture explicitly.

Similar approaches based on identifying areas of the worksheet that can be abstracted as a single object have been widely used. Erwig [2009] applies this approach to reasoning about the structure of the worksheet and implementing unit checking. Enhanced visualisations of spreadsheets, to aid comprehension and debugging, are another application. Hermans et al. [2011] present a tool which allows data flows within a spreadsheet to be visualised at multiple levels: global, worksheet and formula. They use the same kind of inferential method to parse the worksheet into blocks of cells that fulfil a common role. Similarly Hodnigg and Pinzger [2015] use this strategy to identify and highlight areas of the sheet that represent a single cognitive unit.

The same general approach may be enhanced if supplementary information is available, as demonstrated by de Vos et al. [2017]. They combine the structural information with an external vocabulary suited to the domain area, in their case for spreadsheets with laboratory or field measurements. Quantities such as scientific units when identified on the sheet can assist in extracting its semantics.

What if cells that the user intended to be part of the same semantic class (in Mittermeir and Clermont's terminology) fail to be detected as such due to formula errors? Dou et al. [2016] have developed a technique for detecting formula inconsistencies in cells that look as if they should be treated as a single piece with their neighbours. Continuing in the line of Hermans' code

smell work, they refer to these as smelly cell arrays. They have developed an automated approach for repairing them.

The lish aims to capture more of the structure explicitly, so that the machine can respect the integrity of collections of cells that form a single cognitive unit and identify such collections in a deterministic rather than inferential way. Instead of the user relying on spatial layout alone and on the unwritten spreadsheet conventions of how related cell areas are arranged, the user is invited to do the arranging as part of structuring their data in the first place.

2.3.4 The formula model

A defining feature of the spreadsheet is that the user may set the value of any cell to be given by a formula, which may refer to the values of other cells. For example, if cell C8 contains the formula "=sum(C2:C7)" then the machine will calculate the sum of the values in the range of cells C2 through to C7 and place the result in cell C8. With the exception of array formulae (see below), a formula affects only the value of the cell in which it resides.

This property of the spreadsheet, that the value of a (non-literal) cell is determined only by that cell's own formula, has the benefit of transparency as to how any particular cell was calculated. But it also gives rise to a problem: if a whole column of cells is to be operated on in the same way (perhaps, we would like column F to contain twice the value in column E), then all the cells to be calculated must contain their own copy of what is to all intents and purposes the same formula. This contravenes the "Don't Repeat Yourself", or DRY, principle as promulgated by Thomas and Hunt [1999]. It introduces a risk of errors during maintenance – what if only part of a column gets updated, leaving some copies of a superseded formula behind? Indeed, the inferential approaches of the previous section were in large part concerned with identifying such regions so that departures from the expected replication can be detected. An alternative approach is to change the formula model so that the replication is not required.

One way to achieve this is to allow formulae to be defined per column, rather than per cell. In the commercial sphere, Lotus Improv was an early pioneer of this approach; it separated the concepts of data, views and formulae. By taking the formulae outside the cells it incurred some loss of directness, which may be why this style of working never became mainstream. It does however live on in Quantrix, a current commercial product.

Apple's current spreadsheet product, Numbers, has taken a step in this direction with the use of automatic named ranges, so that a formula may refer to a column by its heading rather than to a cell within that column. The formulae themselves remain cell-based. Numbers has also addressed the issue of demarcating separate tables, by adopting a canvas as a super-container on which table grids are placed. The lish allows greater generality than per-column formulae – they may be defined in various repeating patterns including over a column, column group, table body, or a set of equivalent columns in related tables.

Spreadsheets commonly allow the output of a formula to be an array. The array abstraction represents an aggregation of cells and once again can cut down on formula replication, as well as representing return values that are intrinsically multivalued, such as the inverse of a matrix. Array formulae are available, for example, in Excel, LibreOffice and Gnumeric. However they can be limiting in practice, since the dimensions must be fixed at the time the formula is entered, imposing restrictions on future row and column insertions into the sheet. The *Table* construct in Excel is a more flexible approach that allows a table to be treated as a single, properly demarcated object. Formulae are still replicated, but in a more automated way than would be possible on the basic grid. It is a little paradoxical, though, that the Table has been introduced as a new abstraction, quite separate from the table-like grid that formed the basis of the spreadsheet all along! The pivot table is another multi-cellular abstraction, to which I shall return in section 2.3.6.

Clack and Braine [1997] aim at "updating the spreadsheet computational model, whilst retaining the essence of the spreadsheet user interface". Their approach is to introduce object-oriented and functional programming features to the spreadsheet – workbooks behave like classes, and worksheets like functions. They address the formula replication problem by allowing whole regions of cells to be the l-values of formulae. The formula syntax itself resembles a functional programming language, for example with expressions like "`map sum [...]`" to operate upon sequences of ranges in a single formula. Behind the scenes the lish is doing something a little similar (section 6.5), but the syntax is of vector arithmetic, so the user does not see anything looking like a higher order function.

Instead of trying to eliminate formula replication, Hermans and Van Der Storm [2015] manage its risks by copy-paste tracking. When the user

fills an area with an equivalent formula by means of copying and pasting, the machine remembers which cells were affected. Then, if there is a subsequent change to the formula in any of these cells, it is possible to update the others so that they remain consistent.

Other developments to the formula model add support for data of more than two dimensions. The Hypernumbers spreadsheet of Guthrie and McCrory [2011] aims at engineering out many of the risks of web-based collaborative spreadsheets. Along the way it includes support for 3D formulae in the form of "z-queries" that operate across a "pile" of sheets. These do not appear to be true 3D arrays, however, in the sense that inserting a row on one sheet in the pile would automatically cause a similar insertion in the others. Analytica software [Henrion, 2004] is more of a visual programming language than a direct spreadsheet replacement, but is applicable to many of the same kinds of problems. It uses "intelligent arrays", which may have any number of dimensions, and which may be manipulated as single objects by formulae. The lish, too, supports arrays with any number of dimensions by virtue of its nested structure.

There are some other developments to the formula model which are not directly concerned with addressing the formula replication and dimensional generalisation problems, but nevertheless I mention here briefly as complementary approaches that could be used alongside the lish.

An important step forward in the capability of the formula model comes from Peyton Jones et al. [2003] who designed a specification for supporting user-defined functions within the grid itself (as opposed to using an external language such as VBA). A prototype was later implemented by Sestoft [2013]. Clark and Hellerstein [2017] replace the usual formula language with the ability for cells to execute arbitrary Python scripts. Their implementation allows NumPy arrays on the Python side to be bound to tables and columns in the spreadsheet. Bidirectional formulae [Macedo et al., 2014] allow the user to change the *result* of a formula and have the change back-propagated to a source cell, such that the formula will produce the new result.

Another aspect is the user interface by which formulae are composed. Roast et al. [2018] do not change the calculation model, but have developed a new user interface for interactive construction of formulae. The textual "formula bar" in Excel is replaced with a visual representation that shows the data flow involved in the calculation.

2.3.5 Separated models

The approaches of the previous subsection seek to address some problems with calculation in the spreadsheet by modifying the formula model. An alternative approach is to make a more distinct separation between calculation and data, so that formulae are either expressed only in a designated portion of the worksheet, or removed from it entirely.

Isakowitz et al. [1995] describe an algorithm to decompose a spreadsheet into a schema (described in a formal mathematical language designed for the purpose) and associated data, independently of the spreadsheet's physical structure. The formal language used for the schema does not appear to be intended for end users, however. A more readable solution might be the method of Hermans et al. [2010] who develop a tool to transform a spreadsheet into a UML class diagram.

Such tools are applicable when a user has an existing, perhaps legacy, spreadsheet that they wish to model. In other situations, the user may wish to go in the opposite direction: they may wish to design a model and then have it implemented as a spreadsheet. This enables in-table cell replications to be automated, eliminating the risk of error from that quarter. The Gencel system of Erwig et al. [2005] is one such approach, allowing templates for tables to be themselves defined within a spreadsheet and then expanded. An extension of this work takes the form of ClassSheets [Engels and Erwig, 2005] which allows the templates to be specified in a more object-oriented way. A more recent extension [Mendes and Saraiva, 2017] allows templates to be more expressive to support a wider range of business logic; specifically, it allows an arbitrary number of subcategories to be nested within a class. Jansen and Hermans [2014] propose an alternative based on a visual spreadsheet description language.

Excelsior [Ireson-Paine, 2005] appears to be one of the more industry-focused tools in this domain, having been applied on live commercial projects, and runs within Excel. By defining spreadsheet objects in a modular fashion it overcomes the demarcation problem, and makes resizing of tables easier than in a conventional spreadsheet.

Hermans and van der Storm [2016] separate the calculation from the data completely. The calculation part of the model is expressed in code (the language used is JavaScript) in an IDE which is juxtaposed with the spreadsheet. The accompanying spreadsheet grid is updated live in response to edits to the code. Sarkar et al. [2018] separate calculation and data in a

similar way, but retain the normal formula language within the worksheet. The code part of their system is an editable view onto the formulae, with some syntax extensions to make the formulae more suited to viewing and editing as standalone code. These extensions include range assignments to avoid formula replication, and inline cell naming within the code.

Separated models, then, can reduce or eliminate certain classes of spreadsheet errors. They appear to be particularly powerful for use cases where the model has a simple structure, but involves more than two dimensions and significant data volumes. In that kind of situation, a very concise model can generate a spreadsheet that would be much more laborious to build from scratch. The main drawback appears to be some loss of directness: the user is now creating a *description* of a spreadsheet, rather than the spreadsheet itself. If a more interactive workflow is desired, the user may prefer a more traditional spreadsheet environment.

2.3.6 Relational tables in spreadsheets

One way of working with a spreadsheet that is particularly suited to data-intensive applications is to store the underlying data in a separate relational database. That way, the user may take advantage of the integrity constraints provided by the RDBMS while playing to the strengths of the spreadsheet for flexible calculations and ad hoc queries. A number of tools have been developed to facilitate this style of working.

One such a tool that is routinely provided by spreadsheets is the pivot table. This provides an interactive interface in which the user may specify filtering, aggregation and cross-tabulation queries on some given relational table. The table may come from an external database or simply be stored as a worksheet in the current workbook. Modern versions of Excel have enhanced these capabilities very substantially with the "Power Pivot". This supports more powerful queries, including table joins, and provides data analysis expressions (or DAX) which extend the syntax of normal worksheet formulae to accommodate their application to relational queries.

A number of research efforts have investigated alternative query interfaces. For example, Liu and Jagadish [2009] introduce the SheetMusiq prototype. It is a spreadsheet-like front end to build SQL queries interactively (but without the user seeing actual SQL) onto a backend database. It improves on traditional graphical query builders by expressing the query state in terms of an intermediate (spreadsheet) result that can be directly ma-

nipulated. This allows the user to break their query down into manageable stages and see the intermediate result at each stage.

Bakke and Karger [2016] develop a more advanced direct interaction query constructor, called SIEUFERD. This supports multi-block (nested) SQL queries and provides a visualisation not only of the currently returned data but of the structure of the query itself. This reduces the amount of hidden state. The user can go back and amend a previous stage without losing subsequent stages.

If a dose of relational database discipline can help bring order to the spreadsheet, why not go further and incorporate relational tables within the spreadsheet itself? Cervesato [2007] proposed making relations first class objects on the worksheet, and created a detailed specification for the NEXCEL application. Excel's own Tables construct has evolved in recent versions to support a full relational structure including foreign key relationships.

The table structure within the Morphit spreadsheet [Hawkins et al., 2013] is a kind of hybrid between a relation and a pivot table. It supports lookups and joins between tables, and aggregation and filtering within them. Tyszkiewicz [2010] demonstrates how a fairly general set of SQL queries (including CREATE TABLE statements and table joins) can be implemented solely using native spreadsheet facilities and worksheet formulae. Sadaphule and Shaikh [2016] have implemented a compiler which uses similar techniques to generate a correct spreadsheet from a given source SQL file. The output is a standalone spreadsheet, independent of the compiler. Hence that part of the user's model that is best modelled as a relational schema can be generated automatically entirely within the spreadsheet, without external dependencies.

The user may have something of a dilemma, however, between adopting a more formal approach at the outset, as opposed to using a less structured approach in a regular spreadsheet. Mangano et al. [2011] sum this up nicely:

> "[Database] modeling tools work under the assumption that users know the structure of their information ahead of time, which is not always the case. Sometimes the structure emerges only after the user has entered and manipulated at least some of the information."

They present a prototype where cells in a table can be *freeform* – they just contain the user-entered content as-is, or *managed* – they are updated in

line with an underlying content model. They provide a tool allowing the user to migrate freeform data into managed form.

The lish does not seek to replace the relational model, so is less influenced by this area of the technology. The way in which it captures regularity and repeating patterns may however be more appropriate in certain cases; as noted in the discussion of "tidy" data earlier (subsection 2.1.3), the pure relational form is not necessarily the most readable.

2.3.7 Alternative data structures

The simplicity of the grid of cells, at least in its classical form, does have some disadvantages. It provides only a lightly guided imposition of structure. This leaves the user free to superimpose their own, more complex structures, but with little safety net: data that are not "table shaped" can still be represented, but the logical relationships between their elements are typically not captured, nor is their integrity enforced. And modelling data that are three dimensional (or higher) is not straightforward.

Another problematic aspect is the lack of demarcation between regions of the grid that actually represent separate objects. Not only does this risk formula references "leaking" outside the objects they should be accessing, but if two unrelated tables are placed one above the other on the same grid, they will be forced to share common column widths even though there is no logical reason why their columns should align.

Creating a spreadsheet-like system on top of an underlying structure that is not the grid seems to have been only little investigated. An early exception was the work of Burnett et al. [2001]. They implemented a research prototype, Forms/3, which introduced a range of new abstractions to replace the traditional cell/grid model. A large part of this work concerned extending a spreadsheet-like style of working to graphical programming and real time I/O, which are outside my scope. But of particular interest here are their representations for tabular data. Cells are placed within *dynamic matrices* which themselves are placed within forms; the latter behave visually like a canvas, and organisationally like a module or class definition. Formulae may be defined over an entire dynamic matrix or a region of one, and importantly the size of the matrix may be computed at run time, providing dynamic storage allocation in a spreadsheet setting.

More recently there has been something of a resurgence of interest in this area. Miller and Hermans [2016] propose an alternative model that would

allow a user to migrate a normal spreadsheet into a more structured form; their technique can be applied selectively alongside ordinary cells, providing a "gentle slope". Their model involves capturing extra structure within the sheet margins in the form of *semantic axes*. These can include rows or columns that nest within some larger category, such as quarters within years, or product type within revenue. Formulae may then be defined over a *cell grouping* specified in terms of the axes, avoiding per-cell replication of formulae. The axes can also include *dynamic nodes* which allow the column structure itself to be a live copy taken from elsewhere on the sheet.

Chang and Myers [2016] extend the spreadsheet grid to support hierarchical data, by allowing nested cells. Their GNEISS application allows arbitrarily nested JSON data to be imported without flattening. They extend the formula notation with expressions like "=B1.1" to mean the first nested cell within cell B1, and provide extensive facilities for reshaping and summarising; aggregation functions such as COUNT respect the nested structure.

McCutchen et al. [2016] articulate with great clarity how despite looking like a table, the spreadsheet ironically is not very well suited to operating upon tables! In an alternative model named Object Spreadsheets, they represent each object type in a table, where the rows are instances of that type of object and the columns are its attributes (though the underlying schema is not limited to this choice of representation). Where the non grid-like behaviour comes in is that attributes are allowed to be multivalued, so one "row" may contain nested cells under some of its attributes. Furthermore, an object can *own* one or more other objects, again represented by nested rows; the table representing the owned object type is a column subset of the table representing the owner. Object Spreadsheets also defines an advanced formula language. Formulae are defined per column and accommodate those cases where a column may have multiple levels of structure nested within it. The language also includes object references, which are analogous to foreign keys. The authors make the important observation that:

> "In practical terms, ownership captures the nature of a significant fraction of entity relationships in real applications; for example, if an application includes invoices that contain line items, one would expect deleting an invoice to delete the line items."

That is, a hierarchical model with ownership can represent in a single table many (though not all) of the use cases that would require multiple tables in

a relational database. The lish benefits in an identical way from this use of hierarchy.

Bažant and Maršálková [2018] describe a prototype ("Nezt") for spreadsheet-like interactive functional programming based, like the lish, upon nested lists of cells rather than a grid. Its programming capabilities include support for user-defined functions. The lists operated on are Lisp-like lists, as opposed to tables.

Some common threads running through the works reviewed in this subsection are the benefits of defining formulae over multi-cellular areas, and the need for many use cases to be able to capture hierarchical structures. The problem with introducing hierarchy into a spreadsheet is that at some level, the grid must reappear in order that tables can be represented at all. So (with the exception of Nezt, it seems) they all combine the hierarchy with a grid in some way. In Forms/3 (and also in Hypernumbers, reviewed earlier) the grid is at the lowest level in the structure – a nest of grids. In GNEISS and in Object Spreadsheets, the top level construct is a grid and there is nesting within it – a grid of nests. Object Spreadsheets additionally defines nesting at the marginal level, which then spans the entire table, a property it shares with the semantic axes of Miller and Hermans.

The lish does not explicitly define a grid at any level, but we shall see that by virtue of its templating behaviour it is able to generate both nests of grids and grids of nests, as well as a rich set of repeating cell patterns including the marginal groupings just described, and higher dimensional arrays. On the other hand, the lish has no concept (as yet) of a foreign key, so is limited in its ability to combine multiple relations.

Another issue that the works just cited grapple with, to varying degrees of success, is that the formula language can easily become complicated by the presence of hierarchy. The lish aims to mitigate this by extending vectorisation to deal with hierarchical data. Formulae in a lish are not bound to be per-column, but may take effect over any of the repeating cell patterns defined by the templates.

2.4 The cognitive dimensions of notations

In this section, I move on from descriptions of the technology to a language for its evaluation. The *cognitive dimensions* of Green and Petre [1996] provide a standardised vocabulary for evaluating the usability both of static notations and of programming environments. They define thirteen dimensions

in all, and I shall visit most of them in greater or lesser detail throughout the remainder of this dissertation. Here, I introduce the dimensions briefly[2]. For convenience I have divided them into three categories of my own: structure, visualisation and calculation – though my division is only an approximate one, since some dimensions could apply in more than one category.

2.4.1 The dimensions of structure

In this category I place four of the dimensions that we might associate with the underlying data model. Strictly speaking, the cognitive dimensions do not evaluate an underlying model, but rather a notation – the form in which that model is presented to and manipulated by the user. Nevertheless, when the notation is a "thin" layer over an underlying structure, there are some cognitive dimensions which may be identified as being primarily driven by that structure:

Abstraction gradient – the extent to which the user must master new concepts in order to progress.

Closeness of mapping – how easy it is to translate the real-world situation into its model representation.

Premature commitment – a need for the user to provide or act on detail that will not be discovered until later in the development process.

Viscosity – the property of a representation, once formed, of being hard to change or amend.

There can be a tension between easing the abstraction gradient and improving closeness of mapping: providing richer levels of abstraction might improve the latter at the expense of the former. Premature commitment in my context would involve demanding of the user an overly detailed specification while they were still in the early stages of model-building, while high viscosity would imply a difficulty in changing the model if they got that specification slightly wrong, or when the requirements changed.

[2]The definitions of the dimensions in this section are paraphrased from Green and Petre [1996].

2.4.2 The dimensions of visualisation

The lish as a structure is not bound to a single visualisation. But since it is intended to support interactive, spreadsheet-like use, at least one visualisation must be defined for this purpose. Some relevant cognitive dimensions are:

Diffuseness – verbosity; or more generally, a use of symbols or display space that is disproportionate to the information conveyed.

Hidden dependencies – relationships between different parts of a system that are not apparent to inspection.

Role expressiveness – the extent to which the user can tell by looking at the representation what each part is for.

Secondary notation – annotation or formatting supplied by the user which is not germane to the representation itself, but adds extra meaning for the user.

Visibility – the extent to which the model may be viewed "as a whole", and in particular the ability for interdependent parts to be seen juxtaposed.

These are characteristics to do with good visualisation design, and as such are only weakly coupled to the underlying structure itself. However, with regard to secondary notation (annotation and formatting that assist the user but have no meaning to the machine), it could be said that the lish itself "drives" the visualisation. Spreadsheet users frequently use secondary notation by applying formatting to distinguish which parts of the model belong together, or to separate headings from data. In the lish, these aspects are an intrinsic part of the structure, so the visualisation can potentially distinguish them automatically.

2.4.3 The dimensions of calculation

Finally there are some dimensions that have most to do with how the model handles calculations, as expressed by its formula language:

Consistency – the ability for the user who has seen part of the representation to infer more of it.

Error proneness – the extent to which the representation risks careless slips, e.g. from distinct meanings that look deceptively similar.

Hard mental operations – things that are made difficult by the nature of the representation itself, rather than because they are intrinsically so.

Progressive evaluation – the ability to see the result of part of a calculation or program that is under construction.

The role of the vectorised formula language in alleviating hard mental operations will be pertinent here. The spreadsheet already offers the user an excellent environment for progressive evaluation, which the lish inherits.

2.5 Conclusion

In this chapter, I began by reviewing a number of ways of organising data in tables, from low-level representations upwards. I noted in particular the well-established principles of the relational model, and their modern interpretation in the light of "tidy" data. The end user development literature highlighted the importance of features such as programming by example, directness, and immediate visual feedback. In the analytical context, EUD needs to accommodate workflows that are exploratory and incremental, and the ability to treat "results as data".

The spreadsheet remains one of the most successful environments for this purpose, but its accessibility and ease of use come at the expense of making spreadsheet programs vulnerable to errors and difficult to verify for correctness. Approaches that have been investigated to improve this situation have included the use of tools that can infer the structure and identify inconsistencies, and separating the model that generates the spreadsheet from the spreadsheet itself. With the lish I am investigating capturing the model structure more explicitly, to remove the ambiguity associated with inferential methods. And in this project I prefer an integrated over separated model to provide the user with greater directness, while acknowledging that separated views onto the same model can still be beneficial.

I also reviewed the spreadsheet formula model. The large scale replication of formulae in many spreadsheet applications is a violation of the "DRY" principle, and poses risks to correctness and maintainability. I reviewed a number of improvements, which typically define formulae on a per-column basis. The lish seeks to improve flexibility by defining formulae over more general sets of equivalent cells, not just columns. By extending vector arithmetic to accommodate hierarchical structures, I address some

of the difficulties that these structures create with conventional cell formula syntax.

Another approach I reviewed was making the spreadsheet behave more like a relational database. The discipline this enforces on the user is helpful in preserving data integrity and consistency, but reduces flexibility and requires greater user planning in advance (*premature commitment*). My aim in the lish is a structure that is flexible enough for more ad hoc modelling, while allowing the user to capture and enforce structure where it would be helpful to do so.

The underlying spreadsheet grid has remained a constant throughout its history and has received only limited attention by researchers. But recent work, especially that by McCutchen et al. [2016], suggests that applying spreadsheet-like formulae and direct manipulation to structures other than the grid is a promising avenue to explore. In this project, I apply this approach to a very simple base structure, consisting as it does only of nested lists of cells. By introducing a template-forming behaviour (to be described), I allow many richer structures to be represented.

Chapter 3

The "lish" data model: a first sketch

The three chapters that follow this one will develop the lish, and the extensions that support its visualisation and use for calculations, in a formal way. Those chapters will necessarily be detailed and technical.

So it might be helpful to survey the wood before studying the trees. In the present chapter, I introduce the lish informally. I start by jumping ahead, with a preview of what a lish-based model of some spreadsheet-like data will look like. I then give a simplified pictorial representation of how a data model based on lists of cells can be used to model this kind of data.

3.1 Motivating example

Figure 3.1 shows some (fictitious) data on energy consumption for a number of buildings on a university campus. The consumption is broken down by building, quarter and year. The price per megawatt hour is fixed for all buildings within each year, but varies from one year to the next. The table includes both some raw data (the quarterly consumption figures) and some calculated values (the annual totals and costs). Its presentation makes use of a number of visual attributes – shading, grid lines, and the spacing and relative positioning of the text and numbers – to highlight natural groupings and hierarchies within the data.

The figure looks very like a spreadsheet (or a pivot table), and indeed it could easily have been implemented and rendered as such. But the data here have in fact been represented as a lish, and the figure is a screen shot from the prototype lish editor. Although a lish is not the same thing as

Heat & Power - energy consumption monitoring

v3

Quarterly power consumption in MWh by building & year

Fictitious data for test purposes only

Year	Price	Building	Consumed	Q1	Q2	Q3	Q4	Total	Cost
		Perry							
		Venables							
		Wilson							
2016	100	Building	Consumed	Q1	Q2	Q3	Q4	Total	Cost
		Perry		21	10	4	9	44	4400
		Venables		17	9	2	7	35	3500
		Wilson		19	10	3	7	39	3900
2017	103	Building	Consumed	Q1	Q2	Q3	Q4	Total	Cost
		Perry		21	11	4	10	46	4738
		Venables		18	9	3	8	38	3914
		Wilson		20	11	3	7	41	4223
-	-	-		-	-	-	-	-	-

Consumption extract for Venables building

17	9	2	7
18	9	3	8

Figure 3.1: An example of data represented in lish form: fictitious energy consumption data for some buildings on a university campus.

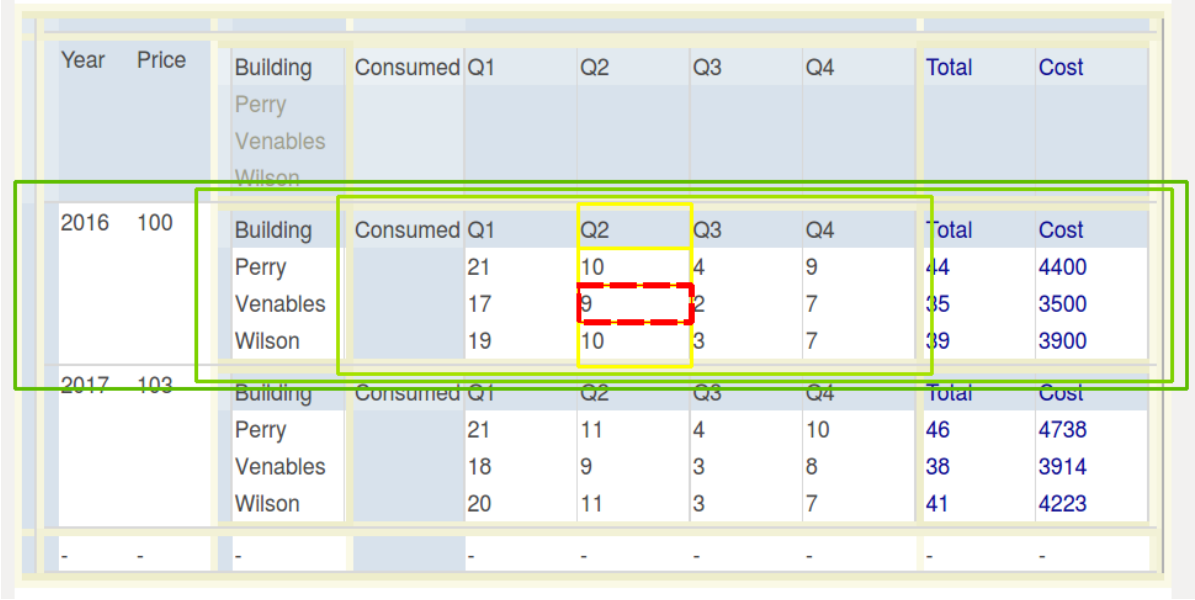


Figure 3.2: Highlighting the nested structure of the data from the previous figure. Note that the frames are part of the screenshot and not an annotation.

a spreadsheet, it can be used to represent many forms of spreadsheet-like data. As stated in the introduction, it is based not upon grids of cells but upon nested lists of cells. In a lish, these lists may be made subject to a "template" behaviour that allows sets of related lists – the columns of a table, for instance – to be kept in alignment. The nested frames in Figure 3.2 show some of this structure more explicitly: the inner grid reveals that the local "Q2" column is a list of four cells, which resides within a parent list labelled "Consumed", which in turn is part of a grandparent labelled "Building", and so on. Those frames can be switched on at will in the editor (they are part of the screenshot, not an annotation) to help the user see what the underlying structure is.

Why might a data analyst choose to represent their data in lish form? There are several potential advantages, all of which stem in some way from the property that the lish expresses more of the structure of the model than a simple grid of cells would do.

One manifestation of this extra structure can be seen in Figure 3.1. The heavy dashed border around the cell at the first intersection of "Venables" and "Consumed" is a cell cursor, which can be moved around just as in a spreadsheet. The bluish-grey shading of this cell identifies it as a *template cell*. The editor interprets template selections specially: in this instance, it means "give me all the cells relating to 'Consumed' and 'Venables', in all of

Year	Building	Quarter	Price	Consumed
2016	Perry	Q1	100	21
2016	Perry	Q2	100	10
2016	Perry	Q3	100	4
2016	Perry	Q4	100	9
2016	Venables	Q1	100	17
2016	Venables	Q2	100	9
2016	Venables	Q3	100	2
2016	Venables	Q4	100	7
2016	Wilson	Q1	100	19
2016	Wilson	Q2	100	10

Figure 3.3: The example lish data in long form

the years". The lighter borders around the three rows of cells answering this description sprang up automatically when the user navigated to the given cursor position. They form an implicit selection, which may be extracted in compact matrix form for calculation or viewing – as has been done in the small table at the bottom of the figure.

The upshot for the user is that both navigation and calculation can be carried out, not by referring to arbitrary cell ranges, but by referring to logically coherent subsets of cells that describe some feature of the model: a column, a table, a set of related rows, or (as we shall see later) some more complex pattern. Let us turn momentarily to the long or "tidy" form of the same data in Figure 3.3: this is the form that databases and many data science tools would use. There is a one-to-one correspondence between the grey template cells of the lish, and many simple filters that we might wish to employ on the long data – for example, all the consumption values relating to Q2, or the entire price column.

These properties of the lish may be applied to good effect when it comes to calculation. The lish supports a formula-based calculation model similar to that found in the spreadsheet, but borrows the idea of "vectorisation" from data science languages like R – with some extensions to accommodate the nesting of vectors within vectors. Formulae are typically, though not necessarily, defined on a template cell and take arguments that are other template cells. Hence a single formula may both operate upon and populate some range of cells, which need not all be adjacent.

The machine knows what the structure is, and this is information it can apply to the service of the user. Take for example the "Total" column. A single formula (located in the first "Total" cell) provides for the whole column, and it is set simply to be the sum of the "Consumed" table. There was no need for the user anywhere to ask explicitly for row sums, since that was deducible from the related structures of the input and output ranges. Similar logic allowed the "Cost" column to be expressed merely as the product of "Price" and "Total", with the machine safely entrusted to determine which of the two prices belonged to which of the six totals.

Now let us look at what would happen if the user were to create an extra building, by inserting an extra row below any of the cells containing "Venables". The lish editor would respond by inserting rows, not only here, but also below each of the other two "Venables" cells. Consistency is maintained, and no modification is required to the original formulae either. A related idea is that the result of a single cell formula may be a lish of arbitrary size, which becomes nested within that cell. This provides dynamic allocation, relieving the user of the need to pre-allocate sufficient cells for the expected size of a result. The Venables extract at the bottom of Figure 3.1 is an example of such a formula. In a nutshell, when the user edits a spreadsheet, the cell values are recalculated automatically; when they edit a lish, the structure itself may be recalculated automatically as well.

This small example shows some benefits that might be realised by building a spreadsheet-like calculating machine on top of a data model that isn't the traditional grid. Of course, these benefits might come at the cost of making other aspects more difficult, something that I will evaluate in Chapters 7 and 8. But first, how does it work?

3.2 Tables as lists of lists

In this section I sketch the general idea of lishes with the help of a pictorial representation. Figure 3.4 shows the basic idea of building tables from cells and lists. It highlights a key issue that was alluded to in the introduction: we need to impose some constraint to ensure that lists that are supposed to be representing a table behave in a table-like way. For example, we need every row to be the same length and we need all the rows to be kept synchronised if a cell is inserted into one of them. The solution I have adopted is the "template rule" shown in panel (6) of the figure. This allows the user to define a structure that implicitly requires certain lists to be kept

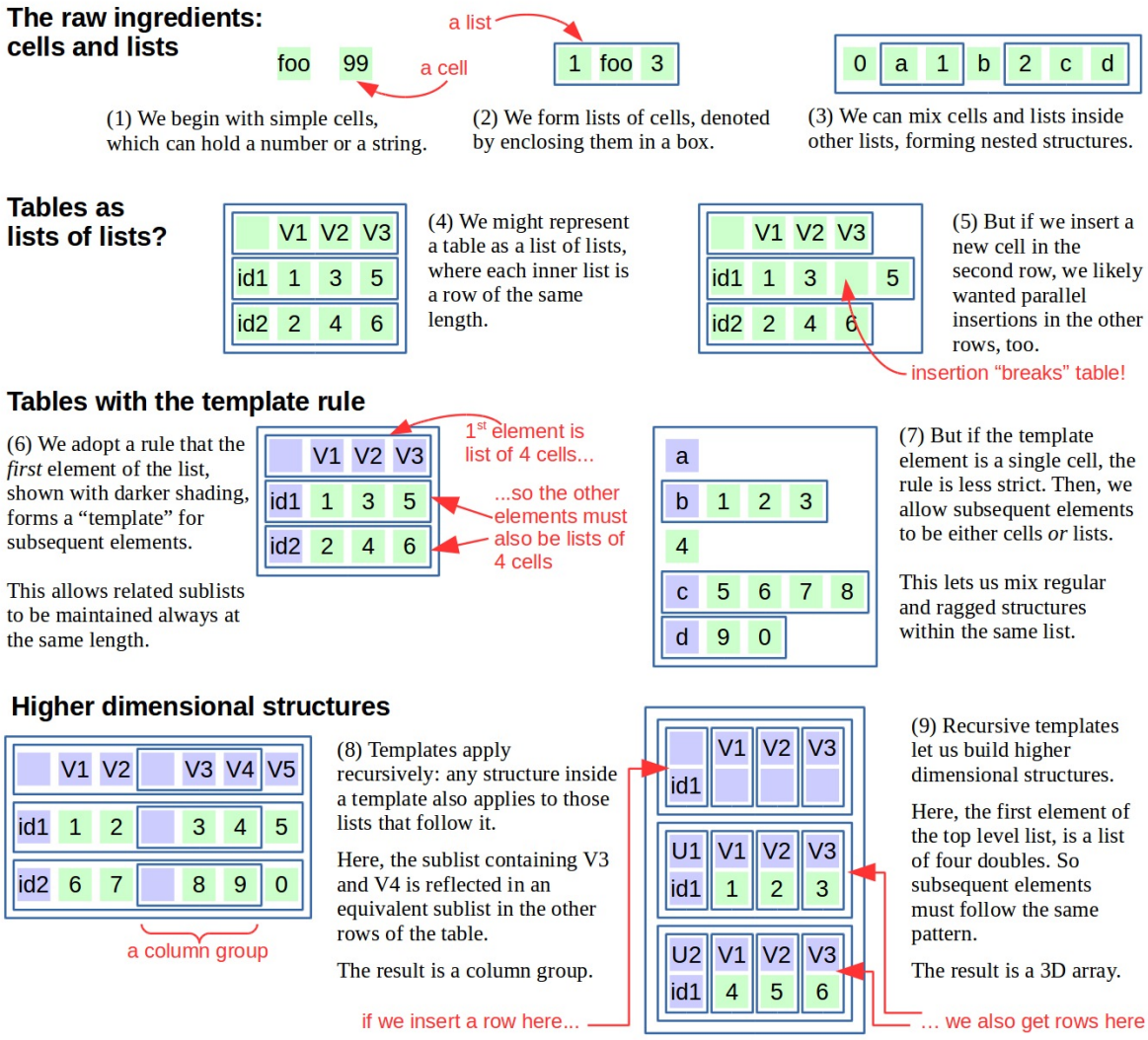


Figure 3.4: The lish in pictures

synchronised, and the machine to respect the implied tabular constraints accordingly. The relaxation in panel (7) adds some necessary flexibility. It allows for the representation of structures that are mixtures of independent cells, one-dimensional lists, and two-dimensional tables. When combined with recursive templates it allows more interesting structures that are not pure N-dimensional arrays.

Panels (8) and (9) of the figure show some simple recursive applications of the template rule. The recursive interpretation applies in two senses. First, any structure inside a template must also appear in those lists for which it is the template. Second, the rule applies inside the templates

themselves. So in panel (9), the template of the top level structure is a 4×2 list (i.e. a list of length four, in which each item is a list of length two). Inside it, the "template of the template" is a list of two cells – a blank, followed by *id1*. This constrains the other lists inside, beginning with V1, V2 and V3 respectively, to be lists of two also.

The column groups of panel (8) are a useful construct when the user wants to make a calculation on some, but not all, of the columns within a table. In the heat and power example, the consumption data were arranged as a column group. This enables formulae to refer to the whole group, rather than Q1 to Q4 individually. The list in panel (8) containing V3 and V4 is legal given that the first element of its parent list is a single cell, by the relaxation of panel (7). The structure of the top row then gets repeated in all the other rows.

3.3 The problem of inconsistent templates

The table in panel (8) of Figure 3.4 contained a column group. It could easily have been arranged to contain a row group instead, by forming a list at the location of cell *id1*, say. Now let us consider what would happen if we wanted both a row and a column group. Attempting to combine the two approaches, we might obtain the structure of Figure 3.5. But this structure highlights a problem. The region of the table containing the values 7, 1, 6 and 8, 2, 7 apparently needs to be of the same structure as two different templates, of conflicting lengths. The structure as shown in the figure looks like a pragmatic compromise: the region which had a template of length three and another template of length four becomes a 3×4 list. But of course this does *not* obey the rules as set out in Figure 3.4, which make no provision for forming a Cartesian product of two templates like this.

How might we amend the rules to accommodate grouped tables of this nature? Two simple alternative rules present themselves as obvious candidates, but neither will turn out to be quite sufficient:

(1) Whenever an item has two *separate* templates that are lists, of lengths m and n respectively, the structure of that item should be an $m \times n$ list.

(2) As for (1), except for the case $m = n$ where there is no contention. In this case, the structure of the item is merely a list of length m.

The table in Figure 3.5 obeys option (1). But consider panel (9) of Figure 3.4. The sublist with the two cells V1, 1 also has two templates:

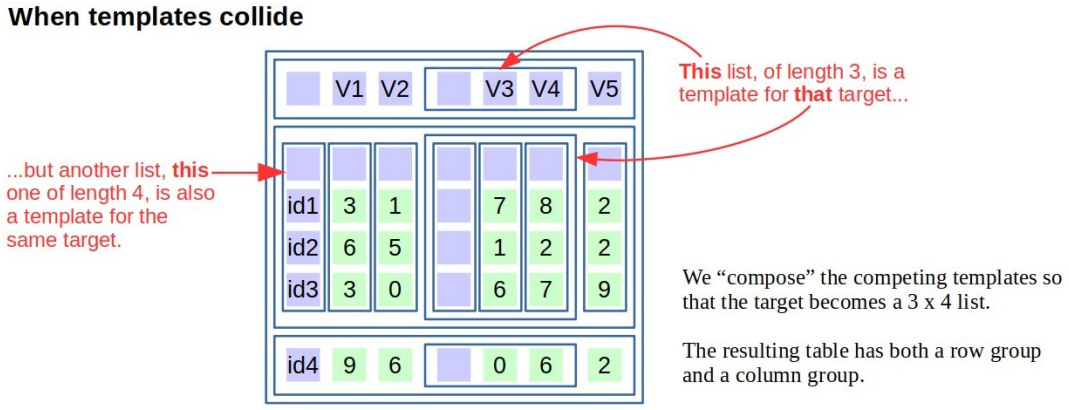


Figure 3.5: Resolving contention between incompatible templates

the V1 sublist above it, and the U1 sublist to its left. In this case however we would *not* want to combine the two templates into a 2×2 list – the number of sublists in the figure would "explode" and the intended 3D array interpretation would be lost. So in amending the rules to allow one type of structure that is of interest, we would be disallowing another.

This problem does not immediately arise with option (2), because in a 3D array we would always, by construction, have the $m = n$ case, so the unwanted combining of templates would be suppressed. But suppose in the table with grouped rows and columns we had, by coincidence, a row group and a column group with an equal number of items. The $m = n$ case would then arise accidentally and the intended combined structure would dissolve into a one dimensional list.

The key to an amended template rule that accommodates both use cases is to notice that for the 3D array, the $m = n$ case was *not* a coincidence, because both templates involved themselves had a structure originating with the first *id1* list at the top left of the table. So the amended rule, informally speaking, is that when an item has two (or more) templates that can be traced to a common origin, then those templates are not to be combined into their Cartesian product; but if the templates had an independent origin, then they are so combined. More complicated cases can arise when the template itself contains sublists, of which some had a common origin and some did not.

In the next chapter, I shall develop a more precise version of the informal description just given. In particular, I shall define formally the process for constructing a template in a manner that captures the intuition, that

structures having independent origin are to be combined, but structures having common origin are not.

Chapter 4

A formal definition of the lish

4.1 Some terminology and notation

4.1.1 Atoms and non-empty lists

My starting point is the *list* in its usual computer science sense of an ordered sequence of elements. All *lishes* will be lists, but not vice versa. The lists of interest will be those whose elements are either atoms, further lists, or a mixture of the two. An atom (for now) may be either a number, a string, or *null*. I shall represent lists enclosed in square brackets with their elements separated by commas. Strings will be shown unquoted where there would be no ambiguity, and *nulls* will be represented by bullets. Some example lists are:

a. [1, 2, 3] – a list of three numbers.

b. [hello, 100, 99, goodbye] – a list containing two strings and two numbers.

c. [“hello, world”] – a list with a single string element (containing a comma).

d. [1, [2, 3], 4] – a list containing two atoms and a nested list.

e. [•, 1, •, 2] – a list where two elements are *null*.

The traditional computer science list includes among its instances the empty list, [ ]. However, an empty list will not be useful in the context of the lish, which is going to confer a privileged status on the first element of each list. An empty lish could in principle be accommodated, but doing so would

complicate the theory unnecessarily by requiring it to be handled as a special case at various points. For the remainder of this chapter, therefore, *unless otherwise specified, all lists are required to contain at least one element.*

4.1.2 The object id of a list

The lish theory is agnostic to the in-memory representation of lists. But I shall assume that any concrete list stored in a computer can be identified by an object id, corresponding to its memory location. For those parts of the discussion where identity is important, I extend the notation to label each list with its object id[1]. For example:

a. $[0x0040\ \ 1, 2, 3]$ – a list with three elements, having object id 0x0040.

b. $[0x0080\ \ 1, 2, 3]$ – a *different* list which happens to have the same three elements, but a different object id.

In both the lists above, the object id is not an element of the list (in the notation, it has no comma after it). Notice that I depart here from the style of treatment based on abstract data types[2] in which two lists both containing the elements [1, 2, 3] are by definition indistinguishable: in the discussion that follows, identity rather than content will determine equality.

4.1.3 Parents, sublists and roots

An issue related to object identity is that some of the properties I shall define on lists are dependent on whether the list in question is part of some larger structure. For example, the list $[2, 3]$ might be treated differently depending whether it occurs within $[1, [2, 3], 4, 5]$ or whether it stands alone. More formally, with each list I shall associate either zero or one *parents*. A list with zero parents is a *root* list, and a list with one parent is a *sublist*. For example, in

$$[0x00c0\ \ 1, [0x0100\ \ 2, 3], 4, 5]$$

the list 0x00c0 is a root list, and the list 0x0100 is a sublist having parent list 0x00c0. No list may have more than one parent; there is no provision for sublists to be "shared" across multiple structures.

Similarly, the *parent* of an atom is the list (if any) within which it is contained; an atom which is not an element of some list has no parent.

[1] In the examples, the object id is an arbitrarily generated hexadecimal address – for conciseness, in a rather small address space!

[2] I use the term *abstract data type* here in the sense established by Guttag [1977].

4.1.4 Indexing and the head

I shall refer to an individual element within a list using an integer index in square brackets after the list. Indices begin at zero. For example, let L be the list [apple, orange, pear]. Then $L[0]$ is equal to "apple" and $L[2]$ is equal to "pear". I shall refer to the element at index zero in a list as its *head*. It is legal for an index to be beyond the range implied by the length of the list; the result of indexing in this case is *null*.

4.1.5 Conformance

I introduce *conformance* as a binary relation between two objects, which may be atoms, lists, or one of each. Informally, conformance refers to the the first object having "at least as much structure" as the second, without regard to atomic content. Conformance is therefore not commutative. It is defined as follows.

- Every atom conforms with every other atom, including itself.
- No atom conforms with any list.
- Every list conforms with every atom.
- Let L, M be lists. Then L conforms with M if and only if:
  - L and M are the same length, n, say; and
  - For all indices i, where $0 \leq i < n, L[i]$ conforms with $M[i]$.

For example:

a. 999 conforms with "hello", because they are both atoms (even though of different type).

b. [1] conforms with 999, but not vice versa.

c. [1, 2, 3] conforms with [4, 5, 6].

d. [1, 2, 3] does not conform with [4, 5, 6, 7], because they are different lengths.

e. [1, [2, 2], 3] does conform with [4, 5, 6] (though not vice versa), because the former is allowed to contain additional structure at index 1.

4.2 The definition of the lish (preliminary version)

I can now state formally a preliminary definition for the lish, corresponding to the earlier description in section 3.2. This definition captures the intuition of the first item in each lish defining a minimum structure, but has yet to address the problem of template "collisions" raised in section 3.3. The latter will be dealt with in section 4.3 below.

4.2.1 The preliminary definition

A list L is defined to be a *lish* if and only if:

- L contains at least one element.
- Every element of L is either an atom or a list.
- Every element of L conforms with the head of L.
- Any element of L that is a sublist, is a lish.

Note that trivially, the head of L conforms with itself.

4.2.2 Example: a list of atoms

Consider the list [1, 2, 3], with head 1. Taking each criterion of the definition in turn, we find: it contains three elements; all of them are atoms; all conform with the head (since any atom conforms with any other atom); and there are no sublists to verify. Therefore this example is a lish.

4.2.3 Example: a sublist is present

Consider the list [1, 2, [3, 4]]. Once again there are three elements; this time there are two atoms and one sublist; and all conform with the head. It only remains to enquire whether the sublist, [3, 4] is a lish. By similar reasoning to the first example, we see that it is. Therefore the list [1, 2, [3, 4]] is a lish. In this example, we have applied the definition recursively.

4.2.4 Example: a non-conforming element

Consider the list [[1, 2, 3], [4, 5, 6, 7]]. This can be immediately rejected as a lish because its second element (a sublist of four) does not conform with its head (a sublist of three).

4.2.5 Example: nested sublists

When a list contains several levels of nesting, a corresponding level of recursion will be involved in establishing whether that list is a lish. Consider the list

$$[[[1,2],[3,4]],[[5,6],[7,8]]].$$

We can verify that this is a lish. I elide the complete verification but note that among other comparisons made along the way, we find that [7, 8] conforms with [5, 6]; and [[5, 6], [7, 8]] conforms with [[1, 2], [3, 4]].

4.3 Keeping track of structure

I now extend the preliminary definition to accommodate the situation of section 3.3. In cases like these, we no longer necessarily require each sublist to conform to a single previous element; instead, we might require it to conform to some derived structure, obtained from two or more such elements. And as we saw, the way in which the derivation happens is to depend in some way on whether the structure of those elements had a common origin. In order to capture these aspects of the behaviour, I introduce a new abstraction, the trace. This will then lead in section 4.4 to a more precise definition of what is meant by a template.

4.3.1 Traces and archetypes

A *trace* is a list (not necessarily a lish) annotated with an attribute called its *archetype*, which is the object id of an ordinary (non-trace) list. In object oriented language, we would say that the class Trace inherits from List, and adds an additional attribute called 'archetype'. The elements of a trace are either atoms or further traces. A trace must always have the *same length* as the list referenced by its archetype, but may have different internal structure and content. I shall show traces in parentheses instead of square brackets, with the archetype at the end, to distinguish them from ordinary lists. For example, given a list

$$[0\text{x}0040 \;\; 1,2,3]$$

we might have a trace

$$(5,6,7 \;\; 0\text{x}0040)$$

where 0x0040 is both the object id of the first list and the archetype of the trace. In any concrete implementation, a trace must of course have an

object id of its own, which in principle could be shown at the start:

$$(0\text{x}0140 \;\; 5, 6, 7 \;\; 0\text{x}0040)$$

However this will not be needed in practice, so only the shorter form will be used.

4.3.2 Composition: atom with atom

Next I define *composition*, a non-commutative binary operation to be denoted by the $\otimes$ operator. For two atoms a and b:

$$a \otimes b = \begin{cases} a & \text{if } b \text{ is } null, \\ b & \text{otherwise.} \end{cases}$$

For example (using the bullet symbol ($\bullet$) as shorthand for a *null*, as earlier):

a. $1 \otimes 2 = 2$

b. $2 \otimes 1 = 1$

c. $3 \otimes \bullet = 3$

d. $\bullet \otimes 3 = 3$

e. $\bullet \otimes \bullet = \bullet$

4.3.3 Composition: atom with trace

I now extend the definition to allow an atom to be composed with a trace. For an atom a and a trace T of length n having archetype r, $a \otimes T$ is defined to be:

$$(a \otimes T[0], \quad a \otimes T[1], \quad \ldots \quad a \otimes T[n-1] \quad r).$$

That is, we pre-compose a individually with each of the elements of T, and retain the original archetype of T in the result. Similarly, but commuting the operands, $T \otimes a$ is defined to be:

$$(T[0] \otimes a, \quad T[1] \otimes a, \quad \ldots \quad T[n-1] \otimes a \quad r).$$

For example:

a. $9 \otimes (1, \bullet, 3 \;\; 0\text{x}0040) = (1, 9, 3 \;\; 0\text{x}0040)$

b. $(1, \bullet, 3 \;\; 0\text{x}0040) \otimes 9 = (9, 9, 9 \;\; 0\text{x}0040)$

c. $9 \otimes (1, \bullet, (\bullet, 4\ \ 0\text{x}0180), 5\ \ 0\text{x}01\text{c}0)$
$= (1, 9, (9, 4\ \ 0\text{x}0180), 5\ \ 0\text{x}01\text{c}0)$

Note that any trace pre- or post-composed with *null* is unchanged. A similar definition might be used for the case of an atom composed with an ordinary list, but this will not be required here.

4.3.4 Composition: trace with trace

I complete the definition of composition by defining it between two traces. Let T be a trace of length l and archetype r, and let U be a trace of length m and archetype s. For the case where both T and U have the same archetype ($r = s$), their composition $T \otimes U$ is defined to be a trace (V, say) of length l and archetype r, in which

$$V[i] = T[i] \otimes U[i], \qquad 0 \leq i < l.$$

For the case where the archetypes are not the same ($r \neq s$), the composition $T \otimes U$ is defined to be a trace V, again of length l and archetype r, in which

$$V[i] = T[i] \otimes U, \qquad 0 \leq i < l.$$

That is, for the matched archetype case, we compose each element of T with the corresponding element of U, whereas for the unmatched archetype case, we compose each element of T with the whole of U. The latter is the Cartesian product case. The length and archetype of the result always come from the left hand operand, T.

For example, taking a matched archetype case, we have

$$\begin{gathered}(1, \bullet, (3, 4\ \ 0\text{x}0100), 5\ \ 0\text{x}00\text{c}0) \otimes (\bullet, 2, \bullet, 9\ \ 0\text{x}00\text{c}0) \\ = (1, 2, (3, 4\ \ 0\text{x}0100), 9\ \ 0\text{x}00\text{c}0)\end{gathered}$$

and taking an unmatched archetype case, we have

$$\begin{gathered}(1, \bullet, (3, 4\ \ 0\text{x}0100), 5\ \ 0\text{x}00\text{c}0) \otimes (\bullet, \bullet, 9\ \ 0\text{x}0140) \\ = \left(\begin{array}{c} (1, 1, 9\ \ 0\text{x}0140), \\ (\bullet, \bullet, 9\ \ 0\text{x}0140), \\ ((3, 3, 9\ \ 0\text{x}0140), (4, 4, 9\ \ 0\text{x}0140)\ \ 0\text{x}0100), \\ (5, 5, 9\ \ 0\text{x}0140) \\ 0\text{x}00\text{c}0 \end{array}\right)\end{gathered}$$

where the outer list in the result has been displayed vertically, for reasons of space.

4.4 Templates

It is now time to define templates formally. For every list, and every element therein, I shall in fact define two templates: the *prior template* and the *posterior template*. All templates are either atoms or traces, and their construction is determined by the mutually recursive rules below.

The definitions are applicable to any list. If that list happens to be a lish, then there is an intuitive interpretation: the prior template contains some minimum structure with which the element must conform, and the posterior template contains some structure that the element might enforce upon subsequent elements.

4.4.1 The prior template: terminating case

The prior template of any root list is the atom, *null*.

4.4.2 The prior template: general case

Let L be a list of length l with prior template R. For the case when R is an atom:

- The prior template of $L[0]$ is R.
- The prior template of $L[i]$, for $0 < i < l$, is $R \otimes S_0$, where S_0 is the posterior template of $L[0]$.

For the case when R is a trace:

- The prior template of $L[0]$ is $R[0]$.
- The prior template of $L[i]$, for $0 < i < l$, is $R[i] \otimes S_0$, where once again S_0 is the posterior template of $L[0]$.

4.4.3 The posterior template

The posterior template of a list L of length l with prior template R, is the trace

$$(S_0, \quad S_1, \quad \ldots, \quad S_{l-1} \quad r),$$

where

- S_i is the posterior template of $L[i]$, and
- r is the archetype of R, if R is a trace; or the object id of L, otherwise.

Notice that the posterior template of L only takes its archetype from L itself if the prior template of L was atomic. Otherwise, the archetype of the prior is carried over to the posterior.

The posterior template of an atom a with prior template R is $R \otimes a$.

4.5 The definition of the lish (final version)

4.5.1 The definition

I can now give the definition of the lish in its final form. A list L is defined to be a *lish* if and only if:

- L contains at least one element.
- Every element of L is either an atom or a list.
- Every element of L conforms to its prior template.
- Any element of L that is a sublist, is a lish.

Notice that only the third criterion has changed since the preliminary definition: we now evaluate elements for conformance against their *templates* as constructed by the rules of the previous section, instead of only against the *head* of their parent list (though in simple cases, the two are often equivalent).

With regard to the last criterion: when we examine a sublist recursively, all the templates considered must of course be those that apply to the sublist at its given place in the structure. For the avoidance of doubt, we do not evaluate sublists under this criterion as if they were separate root lists.

4.5.2 Example: a list of atoms

Consider the root list $L = [0\text{x}0040 \;\; 1, 2, 3]$. It contains three elements, all of which are atoms. The prior template of L is *null*, so the prior template of the head of L is also *null* and its posterior template is $null \otimes 1 = 1$. We pre-compose this with the prior template of L, obtaining 1 again, to get the prior templates of both 2 and 3. Hence every element has an atomic prior template, with which it conforms. There are no sublists to verify, so we conclude that L is a lish.

4.5.3 Example: the head is a sublist

Consider the root list $L = [0\text{x}0200\ \ [0\text{x}0240\ \ 4, 5], [0\text{x}0280\ \ 6, 7]]$. It contains two elements, both of which are sublists.

We find that the prior template of [4, 5] is *null* and its posterior template is (4, 5 0x0240). Hence the prior template of [6, 7] is also (4, 5 0x0240). So both sublists conform with their prior templates.

It only remains to enquire whether [4, 5] and [6, 7], *with the given prior templates*, are lishes. Observing that the individual atoms within them each have an atomic prior template, we find that they are. We conclude that L is a lish.

4.5.4 Example: a Cartesian product

I now present an example with the same features that first caused all the trouble back in Figure 3.5. Let L be the root list:

$$[[1, [2, 3, 4]], [[5, 6], [[7, 8], [9, 10], [11, 12]]]]$$

which for clarity I show again with selected elements arranged to run vertically:

$$\left[\begin{array}{l} \left[\begin{array}{cccc} 1, & \left[\begin{array}{cccc} 2, & 3, & 4 & \end{array}\right] & \end{array}\right], \\ \left[\begin{array}{cc} \left[\begin{array}{c} 5, \\ 6 \end{array}\right], & \left[\begin{array}{ccc} \left[\begin{array}{c} 7, \\ 8 \end{array}\right], & \left[\begin{array}{c} 9, \\ 10 \end{array}\right], & \left[\begin{array}{c} 11, \\ 12 \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

For the present argument, the orientation of sublists above is *not* significant, and the two forms are to be regarded as equivalent; typesetting a lish in tabular form will be the subject of the next chapter. If we switch to the representation that includes the object ids, we might obtain something like the following (once again, equivalent) list:

$$\left[\begin{array}{c} 0\text{x}04c0 \\ \left[\begin{array}{cccc} 0\text{x}02c0 & 1, & \left[\begin{array}{cccc} 0\text{x}0300 & 2, & 3, & 4 \end{array}\right] \end{array}\right], \\ \left[\begin{array}{cc} 0\text{x}0340 & \left[\begin{array}{c} 0\text{x}0380 \\ 5, \\ 6 \end{array}\right], \end{array}\right. \left[\begin{array}{ccccc} 0\text{x}03c0 & \left[\begin{array}{c} 0\text{x}0400 \\ 7, \\ 8 \end{array}\right], & \left[\begin{array}{c} 0\text{x}0440 \\ 9, \\ 10 \end{array}\right], & \left[\begin{array}{c} 0\text{x}0480 \\ 11, \\ 12 \end{array}\right] & \end{array}\right] \end{array}\right]$$

Now, is the list L above a lish? The first two criteria in the lish definition can be easily verified at all levels. Let us then turn to the remaining criteria: do its elements conform to their prior templates, and are its sublists lishes?

The head of L is the 0x02c0 sublist. Its prior template is *null* with which it trivially conforms, and we can verify by a similar argument to the

previous example that it is a lish. Its posterior template is

$$(1, (2, 3, 4\ \ 0\text{x}0300)\ \ 0\text{x}02\text{c}0).$$

The other element of L is the 0x0340 sublist. Its prior template is the posterior template of the head just found (pre-composed with *null*, which leaves it unchanged). The 0x0340 sublist can be seen to conform to this template. Its elements are two further sublists, which we now examine.

The first one, 0x0380, is a list of two atoms with an atomic prior template(namely 1, the first element of its parent's prior template above). It is therefore a lish. It has posterior template

$$(5, 6\ \ 0\text{x}0380).$$

The second sublist, 0x03c0, is the crux of the matter. For its prior template, we need the $R[i] \otimes S_0$ case of subsection 4.4.2. In this instance, we must compose $(2, 3, 4\ \ 0\text{x}0300)$ (the second element of the parent's prior template, found above) with $(5, 6\ \ 0\text{x}0380)$ (the posterior template of the 0x0380).

This is an example of composition of two traces with *different* archetypes. Referring to the $r \neq s$ case of subsection 4.3.4, we obtain for the prior template of 0x03c0:

$$((2 \otimes 5, 2 \otimes 6\ \ 0\text{x}0380), (3 \otimes 5, 3 \otimes 6\ \ 0\text{x}0380), (4 \otimes 5, 4 \otimes 6\ \ 0\text{x}0380)\ \ 0\text{x}02\text{c}0)$$

$$= ((5, 6\ \ 0\text{x}0380), (5, 6\ \ 0\text{x}0380), (5, 6\ \ 0\text{x}0380)\ \ 0\text{x}02\text{c}0)$$

This is a list of three lists of two, so the 0x03c0 sublist does indeed conform to it.

Verifying the sublists within 0x03c0 is more straightforward, if a little tedious, so I elide most of the details, noting only that when we come to the prior template of the 0x0440 sublist we have this time the $r = s$ case of subsection 4.3.4, and the prior template comes out as $(7, 8\ \ 0\text{x}0380)$. The conclusion is that L is a lish.

4.5.5 Example: a three-dimensional array

We can contrast what happens in a superficially similar example where no Cartesian product is involved. Let L be the root list [[[1, 2], [3, 4]], [[5, 6], [7, 8]]], which could represent a $2 \times 2 \times 2$ array. In long form, L might be:

$$\begin{bmatrix} & 0\text{x}0608 & \\ [\ 0\text{x}0600 & [0\text{x}0500\ \ 1, 2], & [0\text{x}0540\ \ 3, 4]\ ], \\ [\ 0\text{x}0640 & [0\text{x}0580\ \ 5, 6], & [0\text{x}05\text{c}0\ \ 7, 8]\ ] \end{bmatrix}$$

We find that the posterior template of the 0x0500 list is $(1, 2\ \ 0x0500)$, which becomes the prior template for the 0x0540 list. The posterior template of the latter is $(3, 4\ \ 0x0500)$, where the archetype 0x0500 was carried over from the prior template. Hence the posterior template of the 0x0600 list is

$$((1, 2\ \ 0x0500), (3, 4\ \ 0x0500)\ \ 0x0600),$$

and this forms the prior template of the 0x0640 list.

We next examine the head of 0x0640, which is the 0x0580. Its prior template is $(1, 2\ \ 0x0500)$ and its posterior template is $(5, 6\ \ 0x0500)$.

Now we come to the analogue of the scenario that involved a Cartesian product in the previous subsection: what is the prior template of the 0x05c0? We must compose $(3, 4\ \ 0x0500)$ with $(5, 6\ \ 0x0500)$. This time both traces have the same archetype: we have the $r = s$ case of subsection 4.3.4 and no Cartesian product is involved. The resulting prior template is $(5, 6\ \ 0x0500)$.

Validations against the remaining criteria are straightforward, and we conclude that L is a lish.

4.6 Editing a lish

A complicated structure representing various tables does not arrive fully formed: the user of a lish editor will want to build it incrementally, and subsequently modify it to respond to changing requirements or to correct errors. I therefore define some operations that may be carried out upon lishes.

4.6.1 Ordinary list operations

Since all lishes are lists, many of the textbook operations defined upon lists may be carried over. Some of these operations must be constrained, however, to ensure that the result is still a lish. Other operations on ordinary lists if applied unaltered to a lish would typically produce a non-lish result, but can be generalised to provide a lish-specialised version.

Standard list operations that query a list, but do not modify it, are unproblematic for the lish. We can ask the length of a lish, retrieve the element at a given index, and enquire whether a given element is an atom. Modifying the value of a single atom is also safe upon any lish, since this cannot alter the structure of any template, and hence whether any other elements conform to that template.

Creating a new root lish is almost the same as for an ordinary list, with one small change. Traditionally, a newly minted list is empty. That would not be allowed for the lish, so the "create new lish" operation returns a lish containing a single *null*: [•].

Before defining some further operations, I shall require one new piece of terminology.

4.6.2 The archetype of a lish

I have so far defined archetypes (subsection 4.3.1) only on traces, and consequentially upon those templates that are traces. I now extend the definition to a lish:

The *archetype of a lish* is the archetype of its posterior template.

It follows from subsection 4.4.3 that the archetype of a lish whose prior template is an atom is the object id of the lish itself. Otherwise, it will be the same as the archetype of the prior template.

For example, referring back to the lish L of subsection 4.5.5:

$$\left[\begin{array}{c} \text{0x0608} \\ \left[\begin{array}{cccccc} \text{0x0600} & [\text{0x0500} & 1, 2], & [\text{0x0540} & 3, 4] & \end{array}\right], \\ \left[\begin{array}{cccccc} \text{0x0640} & [\text{0x0580} & 5, 6], & [\text{0x05c0} & 7, 8] & \end{array}\right] \end{array}\right]$$

All four inner sublists have archetype 0x0500, which is the same as the object id of the first such sublist, containing [1, 2]. The two mid-level sublists each have archetype 0x0600, which is the object id of the first of those – the one containing [[1, 2], [3, 4]]. Finally the root lish has archetype 0x0608, which is its own object id. So the archetype is in each case the object id of the first in a series of lishes, which establishes a length that some further lishes later on must match (except for the root, which can have no related lishes outside of itself).

The archetype is the key to specialising certain ordinary list operations for the lish, since lish constraints can be preserved by ensuring that those sublists that share a common archetype are operated on consistently.

4.6.3 Specialising for the lish

I now take four operations that can be defined on ordinary lists, and specialise them for use with lishes. The operations are:

insert. Given a list L and an index i, insert a *null* element at $L[i]$. For example, with $i = 2$:

$$[5, 6, [7], 8] \Rightarrow [5, 6, \bullet, [7], 8]$$

delete. Given a list L and an index i, delete the element at $L[i]$. For example, with $i = 2$:

$$[5, 6, [7], 8] \Rightarrow [5, 6, 8]$$

enclose. Given a list L and an index i, increase by one the level of nesting at element $L[i]$. For example, with $i = 2$:

$$[5, 6, [7], 8] \Rightarrow [5, 6, [[7]], 8]$$

Or with $i = 1$:

$$[5, 6, [7], 8] \Rightarrow [5, [6], [7], 8]$$

disclose. Given a list L and an index i, decrease by one the level of nesting at element $L[i]$. It is an error if that element is an atom. For example, with $i = 2$:

$$[5, 6, [7], 8] \Rightarrow [5, 6, 7, 8]$$

In each case, an i that is out-of-bounds is an error.

The specialisation for lishes is in essence very simple: whenever one of the four operations above is invoked upon a lish with archetype A, it is implicitly invoked also upon every other lish that shares archetype A. There are however some additional requirements to ensure that the result of the operation remains a lish:

1. Insertion or deletion at index 0 is potentially problematic, because replacing the head of a lish can have far-reaching effects upon templates downstream. Currently, I side-step this issue by requiring $i > 0$ for these two operations; this also removes any possibility of shrinking a lish to zero length. It would be desirable in future work to relax this restriction provided the result remained a lish.

2. For the insert operation, the newly inserted item must of course conform to its prior template. The example just given specialises for the lish unchanged: the prior template is atomic, so a *null* is still inserted. More generally, the inserted item is a lish having identical structure to the prior template but all its atoms set to *null*.

3. When a sublist $L[i]$ is enclosed or disclosed, it is actually those sublists sharing a common archetype with $L[i]$, rather than the ith elements of those sublists sharing a common archetype with L itself, that must be likewise enclosed or disclosed.

4. A similar variation applies when the $L[i]$ to be enclosed is an atom. This creates a new archetype equal to the object id of the newly created list. The other items to be enclosed are those whose prior template has now acquired this new archetype.

5. Disclosing a sublist that contains more than one element might in some cases produce a result that is not a lish. To avoid this problem I currently require the sublist to have only one element (which might be an inner sublist) before it can be disclosed. Once again, a future version of the lish might be more flexible provided the result remains a lish.

4.6.4 Some examples

Let us continue with the example of subsection 4.6.2, which (omitting the object ids) can be represented as:

$$\begin{bmatrix} [& [1,2], & [3,4] &], \\ [& [5,6], & [7,8] &] \end{bmatrix}$$

We found in subsection 4.6.2 that the four innermost sublists share a common archetype, and the two mid-level sublists (the rows, as laid out above) share a different common archetype.

If we insert an item between 3 and 4, then the other three sublists sharing the archetype of [3, 4] acquire a similar insertion:

$$\begin{bmatrix} [& [1,\bullet,2], & [3,\bullet,4] &], \\ [& [5,\bullet,6], & [7,\bullet,8] &] \end{bmatrix}$$

If we now insert an item between the two "rows", we find that its prior template (omitting the archetype annotations) is

$$((1,\bullet,2),(3,\bullet,4)).$$

Constructing the new item to match this template, we obtain:

$$\begin{bmatrix} [& [1,\bullet,2], & [3,\bullet,4] &], \\ [& [\bullet,\bullet,\bullet], & [\bullet,\bullet,\bullet] &], \\ [& [5,\bullet,6], & [7,\bullet,8] &] \end{bmatrix}$$

Now if we enclose the atom, 4, we create a new archetype. The posterior template of the first "row" becomes

$$((1, \bullet, 2), (3, \bullet, (4))).$$

Constructing prior templates for the remaining sublists, we find for example that 8 has acquired this new archetype in its prior template and hence needs to be enclosed, but 6 does not. We obtain:

$$\begin{bmatrix} [& [1, \bullet, 2], & [3, \bullet, [4]] &], \\ [& [\bullet, \bullet, \bullet], & [\bullet, \bullet, [\bullet]] &], \\ [& [5, \bullet, 6], & [7, \bullet, [8]] &] \end{bmatrix}$$

Three sublists, namely [4], [•] and [8], now share this new archetype; so if we disclose one of them, the other two must also be disclosed. Suppose we have done that, hence reverting to the previous state. Now let us enclose $[1, \bullet, 2]$. This is already a sublist, so the other elements to be enclosed are those sublists that share its archetype. We obtain:

$$\begin{bmatrix} [& [[1, \bullet, 2]], & [[3, \bullet, 4]] &], \\ [& [[\bullet, \bullet, \bullet]], & [[\bullet, \bullet, \bullet]] &], \\ [& [[5, \bullet, 6]], & [[7, \bullet, 8]] &] \end{bmatrix}$$

4.7 Atoms as spreadsheet cells

4.7.1 Splitting the atom

Up until now, atoms have been restricted to a small number of basic types: strings, numbers, and *null*. The intent of the lish however is that atoms should be able to represent spreadsheet cells. This requires a richer type of object that can represent formatting information and formulae, in addition to basic values.

In order to accommodate this requirement, I shall now allow atoms to be hash tables (or associative arrays). The name "lish", incidentally, is simply a portmanteau of "list" and "hash": the lish being a list whose atomic elements are hashes. Each atom will be represented as a collection of key-value pairs, enclosed in braces. For example:

{font_family: Helvetica, formula: "2 + 2", value: 4}

The keys are strings and in the usual way must be unique within each hash table. The order of the key-value pairs is not significant. The values

are basic atoms: numbers, strings or *null*. An atomic hash table, unlike a lish, is allowed to be empty.

The lish itself does not restrict what keys and values may be used. A software application built using the lish can choose its own keys and decide how to interpret them. My prototype lish editor is an example of such an application, and happens to use the keys shown in the example, along with several others. Note that in a slightly confusing choice of naming, I chose "value" as the key of that item in the hash table that is to signify the value to be displayed in the cell. So in the key-value pair "value: 4" above, "value" is the key, and "4" is the value!

The hash tables here support the usual operations of adding and removing key-value pairs, and looking up the value associated with a key. Internally they do not have any template behaviour or other lish specialisms, so all the theory of the lish remains intact now that atoms are allowed to have, as it were, some subatomic structure. There is just one loose end: how are the new hash-like atoms to behave under composition?

4.7.2 Hashes under composition

In the original definition of composition between atoms (subsection 4.3.2), the second atom unless *null* completely replaced the first. I now revise this definition so that when two hashes are composed, the original atomic composition operation takes place piecewise between key-value pairs having the same key.

Let g, h be hashes. Then their composition $f = g \otimes h$ is defined to be a hash with the following contents:

- The set K of keys in f is the union of the set of keys in g with the set of keys in h.
- The value in f associated with a key $k \in K$ is determined as follows:
  - If k occurs in only one of g, h and is paired there with value v, then f contains the pair k, v.
  - If k occurs in both g and h, being paired with values v and w respectively, then f contains the pair $k, v \otimes w$, where the original definition of atomic composition is applied between v and w.

To sum up, whenever a key appears in only one of g and h it is carried over along with its corresponding value into their composition. If the key

appears in both g and h, it will also appear in the composition with a value that is the composition of the separate values from g and h. Note that an empty hash performs the original role of *null* in this context.

For example, if

$$g = \{\text{font\_family: Helvetica, font\_size: 12, value: 999}\}$$

and

$$h = \{\text{font\_family: Times, font\_weight: bold, value: } \textit{null}\}$$

then

$$g \otimes h = \{\text{font\_family: Times, font\_size: 12, font\_weight: bold, value: 999}\}$$

In this example, both hashes contained a font family so the one on the right hand side in the composition "won". The font size was unique to g and the font weight was unique to h, so both were carried over unchanged. Both hashes contained a "value" key so the one in h took priority; but since that was *null* the one in g "showed through".

4.8 Inheritance

The behaviour of hashes under composition allows templates to provide an inheritance behaviour. The prior template of each atom is a hash, which in general will contain a number of keys, propagated from other atoms that occur earlier in the structure. Consider the following lish where, anticipating the next chapter, I am assuming a tabular interpretation.

$$\begin{bmatrix}
\left[\begin{Bmatrix}\text{font_family:}\\ \text{Helvetica}\end{Bmatrix}, & \{\text{value: Name}\}, & \{\text{value: Dept}\}, & \begin{Bmatrix}\text{value: Phone,}\\ \text{font_size: 14}\end{Bmatrix}\right] \\
\left[\{\}, & \{\text{value: Alan}\}, & \{\text{value: Computing}\}, & \{\text{value: 1234}\}\right] \\
\left[\{\}, & \{\text{value: Brenda}\}, & \{\text{value: Chemistry}\}, & \{\text{value: 2345}\}\right] \\
\left[\begin{Bmatrix}\text{font_size:}\\ 16\end{Bmatrix}, & \{\text{value: Charles}\}, & \{\text{value: Chemistry}\}, & \{\text{value: 3456}\}\right] \\
\left[\{\}, & \{\text{value: Davina}\}, & \{\text{value: Physics}\}, & \{\text{value: 4567}\}\right]
\end{bmatrix}$$

In the above lish, the key "font_family" with value "Helvetica" which occurs in the first atom of the first sublist can be shown to be in the prior template of all nineteen of the other atoms. Similarly, the key "font_size" with value 16 at the left hand side of the row for Charles is in the prior template of every other atom in that row – but not in the other rows. The other "font_size" key with value 14 at the top of the Phone column is in the

prior template of all the other atoms in that column, *except* the one in the Charles row, where the order of composition means that the equivalent key with value 16 has overridden it.

This propagation of key-value pairs through templates has an obvious interpretation: we can say that atoms "inherit" properties that are defined elsewhere. In terms of tables, all the cells in a table inherit from the cell in the top left corner, all cells in a column inherit from the top cell in the column, and all cells in a row inherit from the left hand cell of that row. Where a cell could inherit the same property that is defined in more than one of those positions, the property at the innermost level of the nested structure takes precedence. Other more complicated inheritance patterns can arise when the lish contains more than one table, or structure within a table.

As a straightforward extension, we can of course let any cell in the table body have a property just its own by including the appropriate key-value pair in the hash representing that cell. For example, if we wanted Physics to be in bold we could expand its hash representation to {value: Physics, font_weight: bold}. The "bold" property is then in the posterior template of the Physics cell, but not in the prior template of that cell, nor any of the others. The lish editor in fact generates the posterior template of the entire lish in order to decide what to render.

So far I have described inheritance only in how it might be applied to cosmetic properties of a lish being displayed. It has a more substantial application, namely its use in the formula model for the lish. A full description of the formula model will be given in Chapter 6, but it will be useful to provide one more definition here.

4.8.1 The inheritors of an atom

Suppose an atom were to be assigned a key-value pair where the key was *unique* within the entire lish. Then the *inheritors* of that atom are defined to be the set of atoms that would have this unique key-value pair in their prior template. Note that it is not required that the first atom actually does possess such a key: the inheritors are simply those atoms that *would* inherit it, were it to be present. This hypothetical construct identifies all those atoms that could in principle inherit some property from the first atom, regardless of whether they actually do so in any particular case. For example, in the telephone list above, the inheritors of the Phone atom are

all the other cells in that column, even the cell where the font_size property was overridden so that nothing was actually inherited.

The concept of inheritors will be applied to multiple cell selections in the next chapter, and to calculations in the following one.

4.9 Conclusion

In this chapter, I have formalised the definition of the lish, expressed in terms of lists of atoms and sublists; the notion of *conformance* played a key role in describing how one part of the structure may be governed by another part. This led to a preliminary definition in which all lish elements conformed to the head of their parent lish.

I then extended the definition to address the problem of inconsistent templates raised in the previous chapter. To do so I introduced the *trace*, which separates the representation of the template from the representation of the lish itself. A trace has an *archetype* which lets it express the idea of repeating structures having a common origin. By defining composition on traces, I provided the necessary fine control over how templates are combined.

I turned next to basic operations on the lish. These were closely based on the corresponding operations on an ordinary list, but were modified to ensure that lish constraints were respected. I then expanded the role of the atomic elements to enable them to model spreadsheet cells, by using a hash representation to express cell properties beyond a simple numeric or string value.

Finally, I showed how the lish definition gives rise to an inheritance behaviour. In the next chapter, I shall make use of this behaviour in visualising the lish, and in supporting multiple cell selections. In the following chapter, I shall call upon it again when defining calculations over multiple cells.

Chapter 5

The lish in two dimensions

5.1 Lishes as tables

In the previous chapter, some tabular behaviour has emerged in the lish: if we represent a table within a lish such that each row is a sublist with a common archetype, the lish operations (for inserting columns, and so forth) have been seen to produce the results expected under the tabular interpretation. But the underlying representation remained simply a list of elements enclosed in square brackets. In this chapter, I shall present a method for typesetting such lists so that they look like tables. I shall then introduce my prototype editor, which allows the resulting tables to be edited interactively.

Typesetting a lish is one of a class of problems where content is modelled as a series of rectangular boxes, which then have to be arranged in a two dimensional space according to certain constraints. The approach is not dissimilar to the one taken by a GUI layout engine, such as GTK or Qt, when it decides how to pack a collection of widgets in a window. A web client rendering HTML/CSS would be another example. In both cases, the objects to be rendered have a specified (but possibly elastic) size, and certain objects have to be vertically or horizontally aligned with others.

A lish in general does not have a straightforward mapping to the various box layouts provided by GUI managers, nor to an HTML table. The way in which lish templates are constructed can result in mixed structures, where some parts are grid-like and other parts are ragged. This can result in cells needing to be kept aligned in the visual representation even when they are not adjacent. Another aspect of the lish that affects the typesetting process is that it prefers structure over markup in determining which elements are

interrelated. With the exception of cosmetic properties such as font size (as described in subsection 4.7.1), the visual layout is determined by parsing the structure, as opposed to inspecting object classes or tags.

One choice in deciding how a table should be represented is whether the innermost sublists should be the rows, or the columns. The lish in fact does not impose one over the other. In particular, the way *inheritors* were defined in the previous chapter means that either way around, cells in the top row have cells in their respective columns as inheritors, and cells in the left hand column have cells in their respective rows as inheritors. The inheritance abstraction has created a symmetry that was not present in ordinary nested lists.

This abstraction is all very well, but if a lish is to be presented visually then clearly each sublist present must be assigned a concrete orientation. Its elements will be arranged next to each other, on the page or the display device, running either from left to right or from top to bottom. Deciding which sublist will run which way is the first task of the geometry manager.

5.2 Horizontal or vertical?

I define here a default decision rule, used by the geometry manager in my implementation, for which sublists are to be oriented horizontally, and which ones vertically. This will turn out to be suitable for many use cases. There is also provision for the user to override the default to obtain a custom layout, by setting an explicit orientation attribute on the first cell in a lish. One constraint is imposed upon the user in this regard: all sublists that share a common archetype must have the same orientation. The procedure used is as follows:

- The root lish is oriented vertically (unless overridden by the user).
- A sublist that is not its own archetype is assigned the same orientation as that archetype.
- For a sublist that *is* its own archetype:
  - The geometry manager first consults any explicit orientation attribute that the user may have set, and assigns this if found.
  - In the absence of such an attribute, the assigned orientation is:
    - orthogonal to its parent's, if it is the first element within its parent;

- parallel to its parent's, if it is not the first element.

For example, consider the following lish:

$$\left[\begin{array}{c} 1, \\ \left[\begin{array}{ccc} [\ 2, & 3, & 4\ ], \\ [\ 5, & [6,7], & 8\ ] \end{array}\right] \end{array}\right]$$

As laid out here, this lish follows the default rules that apply if no user overrides have been set. The root contains two elements, the number 1 and the mid-level sublist containing everything else. It is oriented vertically, so the 1 is placed above the sublist.

The mid-level sublist is its own archetype. It is not the first element of its parent (namely the root), so it is assigned the same orientation as that parent, and it too is vertical. Therefore the inner sublists are also arranged one above the other.

The first inner sublist, $[2, 3, 4]$, is its own archetype. This time it *is* the first element of its parent, so it is assigned the orthogonal orientation to that parent and is arranged horizontally. The second inner sublist, $[5, [6, 7], 8]$, is not its own archetype: its archetype is the $[2, 3, 4]$ just examined. So it too is arranged horizontally.

Finally we have $[6, 7]$. It is its own archetype and is not the first element of its parent; so, like that parent, it is arranged horizontally.

5.3 Aligning the cells

5.3.1 A simple table

For a table that is simply a rectangular grid having no internal structure, and in which all the cells are of the same size, no special procedures are necessary to ensure alignment. Once the orientation of sublists has been determined as above, the geometry manager can simply step through each sublist allocating uniformly sized rectangular areas to each cell. Hence it obtains the coordinates at which each individual cell's contents are to be set. For example, the lish

$$\left[\begin{array}{cccc} [\ \bullet, & \text{Var1}, & \text{Var2}, & \text{Var3}\ ], \\ [\ \text{id1}, & 4, & 1, & 9, \ ], \\ [\ \text{id2}, & 8, & 7, & 2, \ ] \end{array}\right]$$

might be set as:

	Var1	Var2	Var3
id1	4	1	9
id2	8	7	2

Now recalling the hash representation of subsection 4.7.1, let us suppose the user is allowed to assign a key-value pair {cell_width: w}, where w is a width in (say) centimetres, to specify the width of any cell. So the Var2 cell above might become {value: "Var2", cell_width: 5}. The geometry manager will then make this cell 5 centimetres wide. Furthermore, the other cells in the Var2 column are the *inheritors* (as defined in subsection 4.8.1) of this cell. So as long as the geometry manager consults the inherited values, it will set those cells as 5 centimetres wide also, and once again visual alignment is maintained. A typeset version like the following will result:

	Var1	Var2	Var3
id1	4	1	9
id2	8	7	2

This scheme can break down, however, if the user applies a width specification that is not inherited by all those cells that need to be kept in alignment. For example, if the 5 centimetre width above were applied to the cell containing 7, instead of to the Var2 cell, the bottom row of the table would stagger out of alignment. In order to solve this problem, the geometry manager must identify the widest cell in each column, and provide padding to those cells that are narrower to make them up to the same width.

How might this strategy generalise so as to be applicable to any lish? In the remainder of this section, I shall extend the basic idea as applied to a simple table above in two ways:

- I shall develop the procedure for determining which sets of elements need to be aligned. For a general lish, these sets of elements may be more complicated than a single "column" (which is not a formally defined feature of the lish, in any case).
- I shall generalise the idea of padding to a common width, for when some or all of the elements to be padded are sublists rather than individual cells.

5.3.2 Multiple tables

I shall assume that the geometry manager is able to determine, in an implementation-dependent way, the minimum width and height (that is,

excluding any padding) at which each cell must be displayed. It might do this, for example, by:

- setting every cell to a fixed, constant size;
- consulting a measurement explicitly set by the user on that cell;
- consulting an inherited measurement from some earlier cell; or
- calculating the space occupied by the cell's contents, in the appropriate font.

The prototype editor in fact supports all of the above.

I shall describe the alignment procedure as it applies to the *widths* of items. An exactly analogous procedure[1] applies to their *heights*. First let us consider the example of Figure 5.1, where a single lish contains two independent tables. The user has manually set the width of a cell in the Var2 column, and manually set a larger width on a cell in the Var4 column. For brevity, in the lish representation I use the short form showing only cell contents; we can readily imagine adding a cell_width attribute to those cells that have been manually adjusted.

In this example, we require the cells in the second column of Table B (Var4) to be aligned with each other, but we do not want them to be aligned with the second column of Table A (Var1) which is unconnected with Var4. In the normal spreadsheet grid, this is something of a limitation: the grid over-constrains tables that have been placed one above the other on the same worksheet, forcing them to share common column widths. Just as with the lish editing operations of section 4.6, the solution lies in ensuring consistent treatment of those sublists that share a common archetype.

In Figure 5.1, the parent list of the Var4 cell provides the archetype for those lists beginning in id6, id7 and id8. So provided we pad each cell in each of these lists to match the widest cell at the equivalent position in any other list in the set, Table B will be correctly aligned. None of the sublists in Table A has that archetype, so any padding required on cells within Table A will be calculated completely separately, as required.

There are situations, however, when it is not necessary or desirable to align *all* the sublists that share some common archetype. Consider the lish

[1] In my lish implementation, the distinction is abstracted away by defining the *bore* of a lish as its dimension orthogonal to its orientation, and the *bars* as a sequence of subdivisions expressing the dimension of each element parallel to the orientation. For reporting purposes, it will be clearer to assume a concrete orientation.

Table A

	Var1	Var2	Var3
id1	4	2	9
id2	8	7	2

Table B

	Var4	Var5
id6	1	2
id7	3	4
id8	5	6

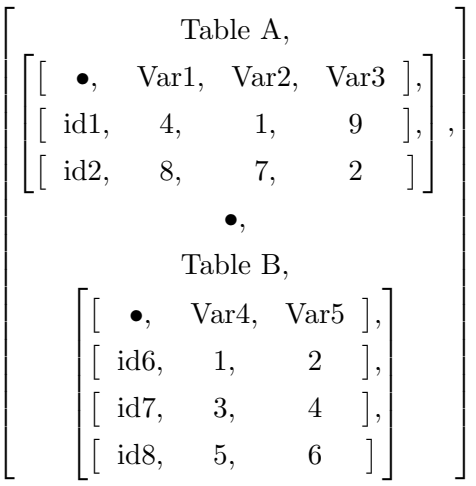


Figure 5.1: Typesetting two independent tables. The lish representation is shown below (omitting the cell_width attributes)

Weather forecast

	9am	12pm	3pm
Narrative			
Temp(oC)			

Monday	9am	12pm	3pm
Narrative	Sunny	Occasional showers	Cloudy
Temp(oC)	12	15	17

Tuesday	9am	12pm	3pm
Narrative	Cloudy	Rain	Cloudy
Temp(oC)	11	10	13

$$\left[\begin{array}{l}\text{Weather forecast,}\\ \left[\begin{array}{l}\left[\begin{array}{llll}\bullet, & \text{9am,} & \text{12pm,} & \text{3pm}\end{array}\right]\\ \left[\begin{array}{llll}\text{Narrative,} & \bullet, & \bullet, & \bullet\end{array}\right]\\ \left[\begin{array}{llll}\text{Temp.,} & \bullet, & \bullet, & \bullet\end{array}\right]\end{array}\right],\\ \left[\begin{array}{l}\left[\begin{array}{llll}\text{Monday,} & \text{9am,} & \text{12pm,} & \text{3pm}\end{array}\right]\\ \left[\begin{array}{llll}\text{Narrative,} & \text{Sunny,} & \text{Occasional showers,} & \text{Cloudy}\end{array}\right]\\ \left[\begin{array}{llll}\text{Temp.,} & \text{12,} & \text{15,} & \text{17}\end{array}\right]\end{array}\right],\\ \left[\begin{array}{l}\left[\begin{array}{llll}\text{Tuesday,} & \text{9am,} & \text{12pm,} & \text{3pm}\end{array}\right]\\ \left[\begin{array}{llll}\text{Narrative,} & \text{Cloudy,} & \text{Rain,} & \text{Cloudy}\end{array}\right]\\ \left[\begin{array}{llll}\text{Temp.,} & \text{11,} & \text{10,} & \text{13}\end{array}\right]\end{array}\right]\end{array}\right]$$

Figure 5.2: Laying out a 3D table. A weather forecast showing (top) the screen representation in the prototype editor and (bottom) the underlying list representation. Note that in the latter, the orientation of the outermost sublist has been transposed for reasons of space. The purpose of the initial 3×4 sublist containing mostly *nulls* will be explained in the next chapter.

of Figure 5.2, which shows a daytime weather forecast split into 3-hourly slots. This time the typeset version shown in the figure is the one provided by the prototype editor. Most of the columns are the same width, but the one for 12pm on Monday has been made wider to accommodate the narrative text, "occasional showers". In this lish, however, all nine of the innermost sublists have a common archetype. So if the geometry manager were to pad every equivalent cell to a common width based on archetype alone, all three of the 12pm columns in the typeset output would share the width of the "occasional showers" cell, which would be wasteful of space. What we would like is to assign the extra width only to the three cells that are actually in the same column as displayed.

5.3.3 Compound indices

In order to refine the criteria for aligning cells, I shall introduce *compound indices*. Within a simple table, such as the Table A of Figure 5.1, we could locate each cell by a pair of indices (j, i) counting its position horizontally and vertically, where $(0, 0)$ is the top left hand cell. For example, the cell

containing the number 8 is at $(1, 2)$. *Compound indices* extend this idea to the nested structure of the lish, and are constructed as follows.

First we obtain the index of the element of interest within its parent. Then, we obtain the index of the parent within the grandparent. We continue in this way tracing the ancestry of the element until we reach the root. For example, the cell with value 5 in the bottom row of Table B in Figure 5.1 is at index 1 in its parent, the lish [id8, 5, 6]. The parent is the fourth row, so at index 3, in the lish that forms the main grid of Table B. Finally this grid itself is at index 4 in the root. We arrange these indices from right to left in a list, and obtain:

$$[4, 3, 1]$$

This list gives a complete description of how to locate the cell with value 5, starting from the root and counting inwards: take element 4 of the root, then element 3 inside that, and finally element 1 inside that.

To obtain a *compound column index*, we filter the above list retaining only those indices where the lish indexed is oriented horizontally. There is only one such lish, namely the innermost one. So the compound column index of this cell is [1].

To obtain a *compound row index*, we apply the complementary filter: we retain only those indices where the lish indexed is oriented vertically. This gives the list [4, 3]. Putting these together, we say that the compound indices of the element 5 are:

$$([1], [4, 3])$$

.

5.3.4 The alignment set

I now define the *horizontal alignment set* of an element X (which may be either an atom or a sublist) within a lish having root R. An element Y in some lish also having root R is in the horizontal alignment set of X if and only if:

- the parent of Y has the same archetype as the parent of X; and
- the compound column index of Y is equal to the compound column index of X.

The *vertical alignment set* is exactly analogous, substituting row for column at the second bullet.

Applying this procedure to the cell containing 5 above, we find its horizontal alignment set consists of the cells containing Var4, 1, 3, and 5 itself: each of their parents shares the archetype of the lish [•, Var4, Var5] and each has a compound row index of [1]. This accords with our intuition that these cells are to be treated as a column of a table; each of them needs to be padded to the width of the widest cell in the set. Cells such as the one containing Var1 are *not* part of this set: although their compound column index is the same, the archetype of their parent is not.

Now let us apply the same procedure to the weather forecast table. The cell containing "occasional showers" has compound column index [1, 2], because its parent and its great grandparent are the only ancestors with a horizontal orientation, and its index positions within those ancestors are 2 and 1 respectively. The cells immediately above and below in the figure likewise have compound column index [1, 2]; these three cells form an alignment set. But now consider the cell containing "rain". Its parent as already noted has the same archetype as the parent of the "occasional showers" cell. But its compound column index is [2, 2], which differs from the latter cell. So it is not part of the same alignment set, as required.

5.3.5 Internal and external dimensions

So far we have only considered examples of alignment sets where every member is an individual cell. But the definition of an alignment set in the previous subsection applies equally when some or all members of the set are sublists. I shall now complete the picture by defining how the dimensions of a sublist, and hence the amount of padding necessary for such cases, are to be determined. Once again, I shall work in terms of widths, but an exactly analogous definition applies to heights.

As already noted, I have assumed that the geometry manager is able to obtain the dimensions of individual cells in an implementation dependent manner. I shall now let the geometry manager accommodate a variable amount of marginal space around the four edges of each lish to be typeset, and will likewise assume that the desired margins on each lish can be obtained by the geometry manager. This will allow for more flexible layouts, since otherwise the cells in adjacent lists would be forced to adjoin directly, leading to a rather dense and cluttered display. I now define *internal widths* and *external widths*, as follows.

- The *internal width*

  - of a *cell*, is its minimum display width, as specified at the start of subsection 5.3.2;
  - of a *horizontal* list, is the sum of the external widths of its elements, plus the widths of its left and right margins;
  - of a *vertical* list, is the maximum of the external widths of its elements, plus the widths of its left and right margins.

- The *external width*, of any element, is the maximum of the internal widths of those items in its horizontal alignment set.

A small example is shown in Figure 5.3. An exaggerated margin is displayed on all the constituent lishes so that the alignment of the various elements may be seen more clearly. We assume that the user has manually set the internal width of the "wide cell" so that it lives up to its description. This cell has one other item in its horizontal alignment set, namely the lish [[foo, baah], [x, [y, z]]] which holds all the other content.

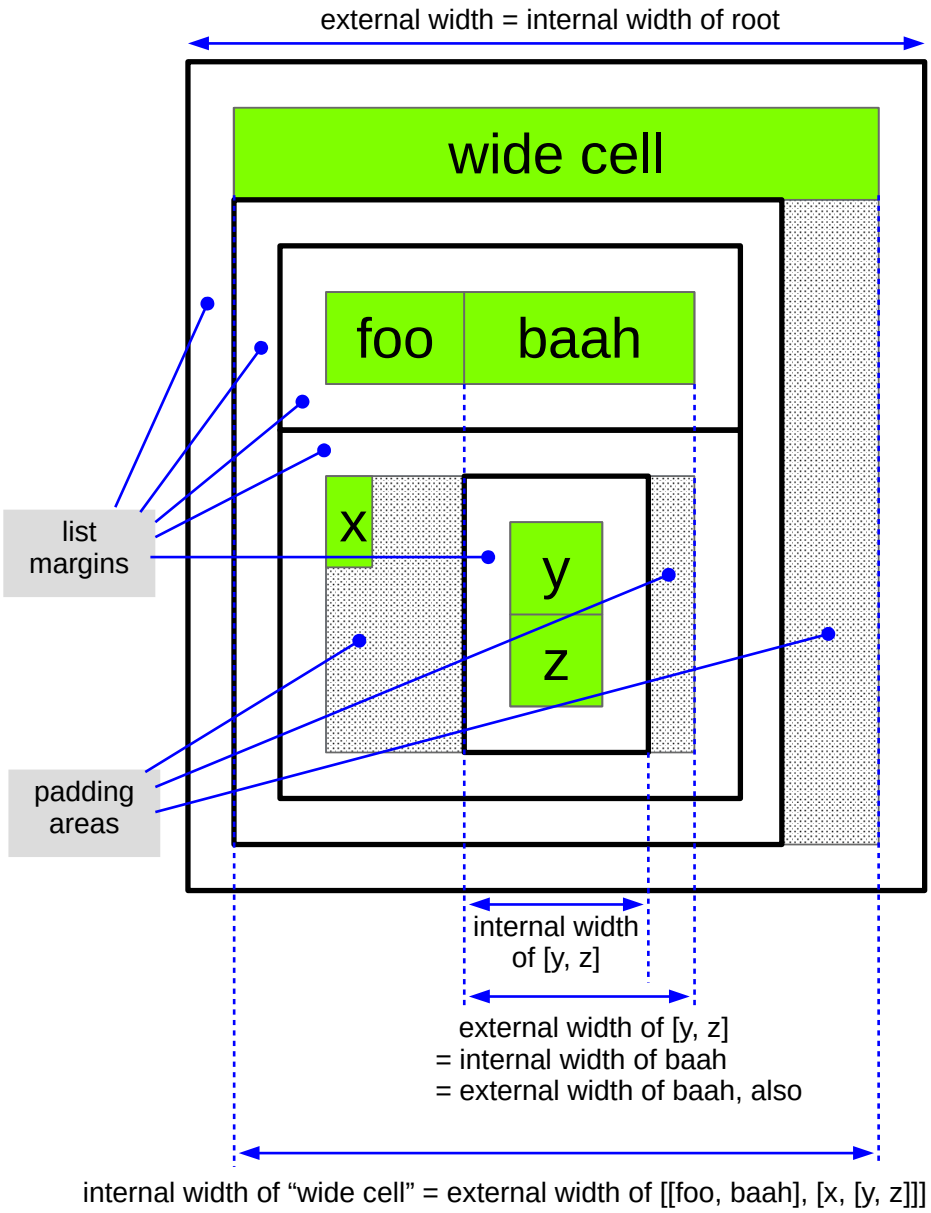
The external width of the wide cell is the greater of that cell's internal width, and the internal width of its sole aligning item. Since the wide cell is the wider of the two, its internal width supplies the external width of them both. The sublist requires horizontal padding, displayed as a stippled area in the figure.

Examining the remaining structure from the inside out, we see that the cells containing foo and x form a horizontal alignment set. Since cell foo has the larger internal width, cell x requires padding to this width, which becomes the external width of both.

We see that cells y and z have equal internal width and are the only members of their horizontal alignment set. So no internal padding is required within the [y, z] list. The external width of each cell is equal to their common internal width. And the internal width of [y, z] is equal to this cell width, plus the widths of its left and right margins. This list, however, is in a horizontal alignment set with cell baah. (Note that it is the *whole* [y, z] list, not its individual elements, that is to be aligned with cell baah.) Since the internal width of [y, z] is less than that of cell baah, the former requires horizontal padding to make the two external widths agree.

The list [y, z] is in vertical alignment with cell x. Therefore the latter receives vertical padding (as well as the horizontal padding just seen) to bring the two to a common external height.

The internal widths of [foo, baah] and [x, [y, z]] now agree, so no further



$$\begin{bmatrix} \text{“wide cell”}, \\ \begin{bmatrix} [\ \text{foo}, & \text{baah}\ ], \\ [\ \text{x}, & \begin{bmatrix} \text{y}, \\ \text{z} \end{bmatrix}\ ] \end{bmatrix} \end{bmatrix}$$

Figure 5.3: Aligning a small example lish. Cells are shown with a green background, and lists framed in black. The internal and external widths are shown on selected elements; in the stippled regions, padding is required. The underlying list representation (not including the cell_width attributes) is shown underneath.

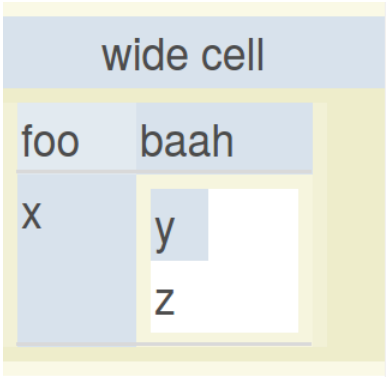


Figure 5.4: The example of Figure 5.3, as typeset by the prototype editor.

padding is required in their combined sublist, [[foo, baah], [x, [y, z]]]. The internal width of this sublist is equal to the sum of the external widths of foo and baah, plus the combined width of the left and right margins.

5.3.6 A typesetting algorithm

The definitions of alignment sets and external dimensions lead easily to a general algorithm for typesetting a lish. I assume that there is a display device or page addressed by coordinates (x, y) in the same units as are used for the widths of cells and margins, and where x increases from left to right and y from top to bottom. I also assume implementation-dependent procedures are available to render the actual content of a cell, and optionally to render any list-level formatting such as shading or grid lines. I state the algorithm in pseudo-code for the horizontal lish case; once again, the generalisation to detect and handle either orientation is straightforward. To render a lish L of length l at coordinates (x, y):

1. Call implementation-dependent procedure (if applicable) to render any background to L, such as shading.
2. Increase x by width of left margin of L.
3. Increase y by height of top margin of L.
4. For i in 0 to $l - 1$:
5. Render $L[i]$ at (x, y).
6. Increase x by the *external width* of $L[i]$.
7. Next i.

8. Call implementation-dependent procedure (if applicable) to render any foreground to L, such as grid lines.

Note that at step 5, the procedure called to do the rendering is the implementation-dependent one if $L[i]$ is a cell, or a recursion into the given algorithm if $L[i]$ is a list.

5.4 A prototype editor

As part of this project, I have built a small prototype "lish editor" which allows a user to enter and edit data in lish form. The editor provides the operations of the previous chapter and displays the resulting lish using the typesetting algorithm above. It also implements the operations of lish calculus defined in the next chapter, so the user may create spreadsheet-like cell formulae and perform calculations. The editor is implemented in Ruby, with the graphics primitives for typesetting (drawing lines, shaded rectangles and text) provided by the Qt library.

For simplicity, the editor is controlled using a command line rather than a GUI, but it has a graphical window to display the lish. Having to type commands for every single operation would be exceedingly laborious, however. Therefore the majority of the commands have been assigned shortcut keys. This gives the editor a somewhat interactive "feel" in use, even though the full GUI conveniences of buttons, menus, etc. are not provided.

5.4.1 Expressing the structure

In this chapter I have covered in some depth how to typeset a lish so that elements of differing widths are aligned in a tabular way: that is, the *geometry* of how a lish is to be presented to the user in two dimensions. The final appearance of the lish as displayed will also depend on other aspects such as the use of gridlines, relative spatial separation, and colour. Some of these could be placed under the direct control of the user by consulting appropriate cell attributes as the lish is rendered, in the same way as was done for fonts earlier. It would be useful, however, if even the default rendition could make effective use of such properties to enhance the way the lish is displayed. Some desirable characteristics of the final output are:

- It should be clear to the user what the underlying list structure is. This addresses the *role expressiveness* cognitive dimension.

- Areas that are intended to be tables should be recognisable to the user as such. This addresses the *closeness of mapping* cognitive dimension.

- The result should be visually engaging (for example, laid out so as to invite easy scanning; free from extraneous clutter).

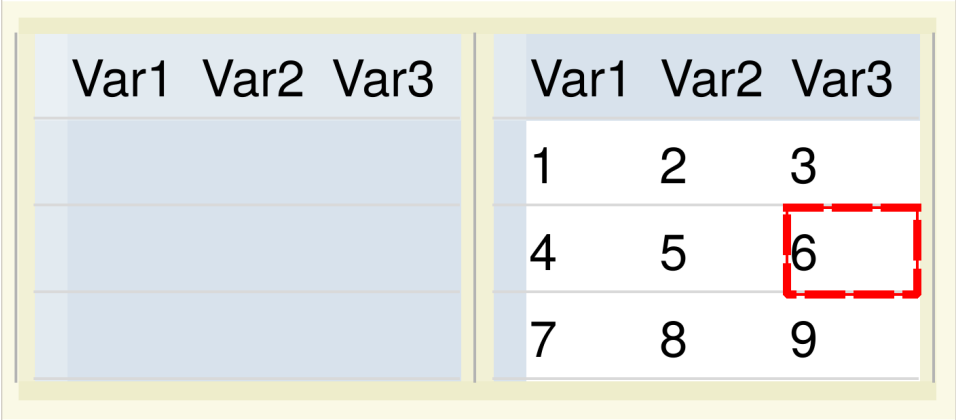
A detailed exploration of these characteristics, and the tensions and trade-offs between them, is beyond my scope here; it would be a large and useful area for future work. But the lish editor has to display *something*! I have therefore made some pragmatic choices that go some way towards meeting the criteria above, and implemented them in the lish editor. In the remainder of this subsection I simply describe these choices without detailed justification (though I do sketch a brief rationale).

Shading and colour

When I first introduced the lish, I used the style of Figure 3.4 on page 48. This style is useful for making clear what the structure is, but does not scale well to more complicated lishes, where an excessive depth of nesting of the boxes would make for difficult reading. The lish editor uses a more subtle indication of the depth of the structure by shading the margins of each lish on a graded scale (I have chosen to use pale brown / creamy shades). To preserve visual differentiation, the scale is recycled once the structure becomes more than four lishes deep. An alternative would have been to use a wider range of shades so as to increase the depth before recycling becomes necessary, but this would give the display a rather dark, "muddy" appearance.

Cells are usually displayed with black text on a white background. As we have seen, however, the first element of each lish has a special status with regard to generating templates. In the next chapter, such elements will acquire a further distinction with regard to how they are used in calculations. So in the editor, I distinguish these template-forming cells by giving them a blue-grey shading. Once again, a fourfold scale is used depending on their depth, and recycled as necessary. Cells that are either themselves the first element of a lish, or lie within an ancestor that is a first element, are treated in this way.

It will be seen that there is a tendency for multi-dimensional lish structures to have rather large numbers of these template-forming cells, many of them empty (represented by *null*). To avoid profligate use of display space,



$$\left[\begin{bmatrix} [\bullet, & \text{Var1}, & \text{Var2}, & \text{Var3}\ ], \\ [\bullet, & \bullet, & \bullet, & \bullet\ ], \\ [\bullet, & \bullet, & \bullet, & \bullet\ ], \\ [\bullet, & \bullet, & \bullet, & \bullet\ ] \end{bmatrix}, \begin{bmatrix} [\bullet, & \text{Var1}, & \text{Var2}, & \text{Var3}\ ], \\ [\bullet, & 1, & 2, & 3\ ], \\ [\bullet, & 4, & 5, & 6\ ], \\ [\bullet, & 7, & 8, & 9\ ] \end{bmatrix}\right]$$

Figure 5.5: A rendition of a lish showing the use of lines and shading (above) and the underlying list representation (below). The red box around the 6 is the cell cursor.

empty template-forming cells are assigned smaller internal dimensions than are given to ordinary cells lying in the body of the lish. In practice I have observed that this scheme is only partially successful – they still occupy more space than is desirable. An improved version of the editor would offer the user the option to hide these cells altogether.

An example of this scheme can be seen in Figure 5.5. All the cells in the left hand half of the table are template-forming (since they all lie in the first sublist of the root), so they have all been assigned the blue-grey shading. In the right hand half of the table, the cells in the top row and left hand column are also template-forming, so they too have this shading; the cells containing the numbers 1 to 9 are not template-forming, so have a white background. The columns that contain only *nulls* have been rendered narrower than the rest, since their cells all have the smaller internal dimensions.

Lish margins

In Figure 5.5, a pale brown margin can be seen around both of the two top-level sublists, and around the root. But no margin is visible around any of the eight innermost sublists. This is because the lish editor has been set to collapse the margin width to zero on certain sublists. The aim is to retain a

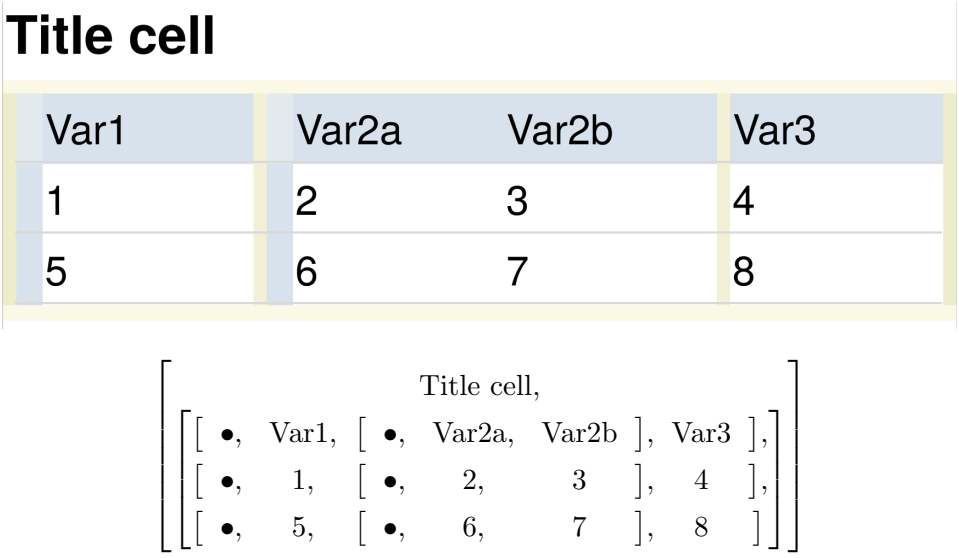


Figure 5.6: The suppression of top and bottom margins around the column group containing Var2a and Var2b.

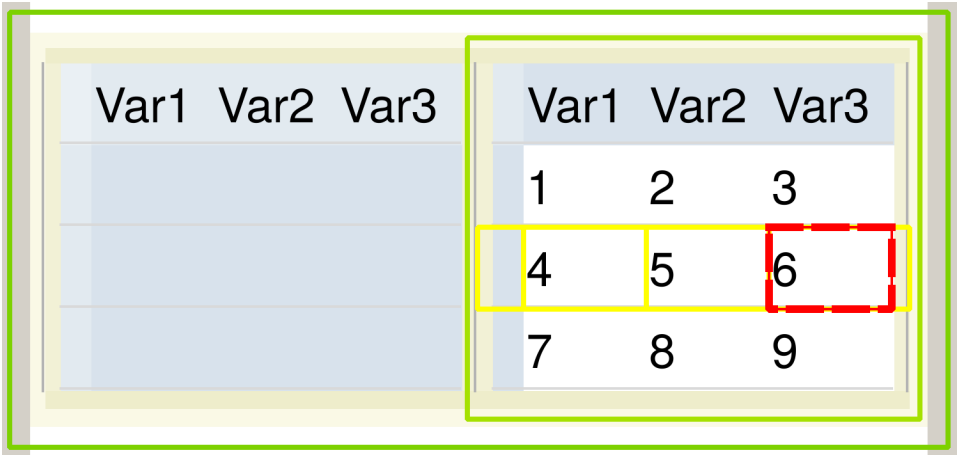


Figure 5.7: The use of a "box" view on the lish of Figure 5.5, to clarify the underlying structure.

margin where the visual separation would be helpful, but suppress it where the result would be an unnecessarily sparse layout (rather like double-spaced text). To be precise, the rules the editor applies are:

- When both a lish and its parent are oriented horizontally, the inner lish is given only left and right margins. A corresponding rule applies when both are vertical. This prevents a spurious increase in the row height within a table when a column group is introduced.

- When a sequence of lishes having the same archetype lie alongside, the margins along their adjacent edges are suppressed. This causes the elements of such lishes to be displayed slightly closer together than would be the case if they had no common archetype relationship.

The first rule can be seen in action in Figure 5.6. The column group containing Var2a and Var2b has a brown shaded margin on either side, delimiting it within the table of which it forms a part. But there is no brown shading between the sublists $[\bullet, 2, 3]$ and $[\bullet, 6, 7]$; the row height is not inflated by the presence of the column group.

Gridlines and box view

The editor uses gridlines sparingly, to mitigate the visual overload of multiply nested boxes. Where a sequence of lishes form the rows of a table, a light horizontal rule is drawn under each, but not between the cells. Similarly, if a sequence of lishes represents columns, a vertical rule is drawn. This gives the user a hint to the underlying structure of the table. A lish that contains a higher than two-dimensional structure is subdivided by slightly heavier rules, examples being the three vertical lines dividing the top level structure in Figure 5.5. Areas containing dynamically generated cells (to be introduced in subsection 6.3.2) are shown with a dotted border.

Early experience with the editor suggested that the rendition of the lish described above, while quite effective as a way of showing lists as tables, tends rather to obscure what the underlying list structure is. To mitigate this problem, I added the option of turning on a "box" view which shows that structure more explicitly. In this view, a full grid is drawn over the parent list of the element selected by the cursor – that is, every element is outlined, giving a "ladder" appearance. In addition, green boxes are drawn around the ancestors of the parent, to a maximum of four levels. These additional annotations give the user a clearer view of the local structure surrounding

the cursor. The green boxes are drawn slightly expanded relative to the sublists that they enclose. This is so that where the margins of nested sublists were suppressed as described above, the boxes will not be coincident, ensuring that an outer box always strictly encloses an inner box. An example is shown in Figure 5.7.

5.4.2 Cursor movement

The lish editor has a cell cursor very similar to the one found in a spreadsheet. The user can move it around using the arrow keys, and commands given to the editor (either typed in the command window, or invoked via the shortcut keys) in general are applied at the current cursor position. One difference from the ordinary spreadsheet cursor is that in the lish, the cursor can be used to select whole sublists as well as individual cells. The user can drill into a selected sublist, taking the cursor inward one level, by pressing Enter, and come back up a level by pressing Backspace. This allows the user to choose at what granularity they wish to navigate the model.

The ragged text problem

In a word processor or text editor, cursor movement has to accommodate the possibility of "ragged" text. If the cursor is in a long line of text with a shorter line adjacent, then a strictly vertical movement of the cursor is not possible: there is no text at all in the target location for it to move to. A spreadsheet never has this problem, because the regular grid ensures there are no "void" spaces. The lish, however, has its own version of the ragged text problem. The user might want to navigate vertically across adjacent sublists of different widths, or horizontally across adjacent sublists of different heights. And the lish adds an extra complication: an adjacent sublist might be of insufficient *depth* to allow like-for-like positioning.

Text editors have adopted an almost universal behaviour for handling this problem. Suppose the cursor is in column 40 and the user presses the up arrow, but the line above is only 20 characters long. This upward movement is not prevented, and the cursor will simply be placed at the end of the shorter line – that being the closest it can get to strict vertical movement. However, the cursor will "remember" that it was in column 40 when the movement was initiated. If it is subsequently moved to another line that does have 40 or more characters, it will return to column 40. So the horizontal position is maintained as far as possible. A particularly important

feature is that if the user presses the up arrow followed by the down arrow, they will always get back to where they started. This contributes to avoiding *premature commitment* by making a navigational misstep readily reversible.

The shadow and residual cursor positions

The lish solution follows the spirit of the ordinary text editor, but requires the cursor to "remember" more than a single column position. In a lish, the cursor position is maintained internally as a pair of compound indices (subsection 5.3.3) comprising its compound column and compound row indices. These indices always reference an actual location in the lish, and are where the cursor is drawn. In addition to these, the editor maintains two other pairs of indices.

First, there are the *shadow* compound indices. Their role is directly analogous to the column "memory" of the simple text editor above. When the cursor is initialised, and immediately after an edit, the shadow indices are set to be the same as the the actual indices. They can be thought of as giving a position where the cursor would "like" to be, but cannot attain in all cases if certain sublists are either not sufficiently long, or not sufficiently deep.

Second, there are the *residual* compound indices. Their role is to remember the location of a nested item when the user has visited it, but then expressly drilled out again. Since the user has now signified that they do not want the cursor at the deeper level right now, the editor should make no attempt to place it there: this is the distinction from the shadow position above. But if the user does subsequently drill back in, the editor consults the residual compound indices to return the cursor to its previous position. These indices are specified *relative* to the shadow position, and are initially empty.

An example of moving the cursor

An example will make clearer the interplay between the actual, shadow and residual positions. Suppose the user has just edited cell Var2a in Figure 5.6. Following the procedure of subsection 5.3.3, we find that the cursor is now at compound indices ([2, 1], [1, 0]). After the edit, the shadow position is equal to the actual position, and the residual compound indices are empty: ([ ], [ ]).

Now suppose the user drills out one level. This is an explicit, not forced,

outward movement, so the shadow follows the cursor and both arrive at compound indices ([2], [1, 0]). The cursor now selects the whole of the [•, Var2a, Var2b] sublist. The residual compound indices become ([1], [ ]), where the index of 1, that was dropped from the compound column index of the cursor upon drilling out, has now appeared in the residual. This index is retaining a memory of where to go should the user subsequently decide to drill back in.

Next, let us have the user press the up arrow. The cursor now needs to move to the single cell, "Title cell", which is the only element available on the row above the current position. So the cursor position moves from ([2], [1, 0]) to ([ ], [0]). The shadow, on the other hand, is now set to the position where the cursor *would* have gone, had the structure on the row above matched that at its previous position: so the shadow position becomes ([2], [0, 0]). The residual position remains unchanged at ([1], [ ]) because the change in depth was entirely forced.

Now suppose the user retraces their previous step by pressing the down arrow. The first thing the editor does is to move the cursor directly down one row, from ([ ], [0]) to ([ ], [1]). If this were the end of the story, the cursor would now select the whole of the sublist forming the main table. But to finalise the new cursor position, the editor consults the shadow and "tops up" those indices where the structure was previously deficient to attain the shadow position, but has become newly available. Hence both the cursor and shadow positions arrive back at ([2], [1, 0]). Once again, no explicit change of level took place, so the residual position is unchanged at ([1], [ ]).

Finally the user might drill back in by pressing Enter. Clearly this must select one of the cells in the currently selected sublist [•, Var2a, Var2b], but which one? If the residual compound coordinates were empty, the editor would simply select the first cell by default, and the cursor would go to ([2, 0], [1, 0]). But since a residual is available, it is consulted to fill in the new final index. Therefore the cursor is located at ([2, 1], [1, 0]), where it began this example, and the residual position becomes empty.

An *en passant* evaluation

Evaluating the "shadow" behaviour of the cursor was not an explicit goal of the user study (Chapter 8). But during that study, no participant had to ask, "How do I get to that cell?", or even commented on the cursor behaviour at all. By appeal to the dog that did not bark, I conclude that the rather

complicated-looking behaviour specified above translates to something that is at least reasonably intuitive for the user.

5.4.3 Implicit cell selections

The cursor in the lish editor has one more talent at its disposal. As we have seen, it normally selects either a single cell, or a sublist. But if the user places it in a *template-forming* cell, it implicitly selects all the *inheritors* (subsection 4.8.1) of that cell, as well.

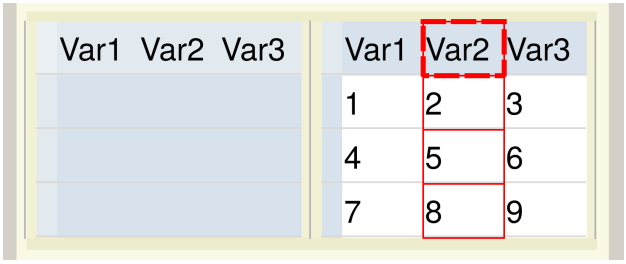
As described in the example of section 4.8, one of the purposes of inheritance is to set formatting properties over a whole row or column, or larger area. The implicit selection behaviour enables a workflow where the user can locate a cell whose inheritors form a compound object (such as a column, or a complete table) that they wish to work on, and then set attributes on that cell that will be inherited by the object as a whole. The implicit selection shows automatically ahead of time which cells will be affected[2]. The same style of working will become more important still in the next chapter, where inheritance will be used to effect calculations on whole objects rather than individual cells.

We have already seen that cells in the row and column margins have as inheritors their respective rows and columns, and the top left hand cell of a table is inherited by the whole table. Some examples of other possibilities are shown in the 3D table of Figure 5.8. We can visualise this as a stack of planes, each of which is a normal 2D table. In addition to any of the selections on individual tables we can select, say, the second row of every table; or a one-dimensional list that "drills through" the stack of planes collecting one element from each. Similarly, if we had a family of irregular structures, possible selections would include any individual member of that family, and a set of items collected from equivalent positions within each member.

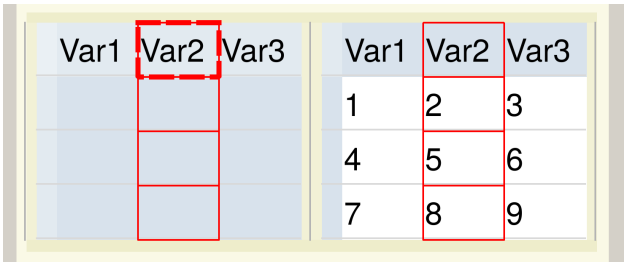
5.5 Conclusion

This chapter marks further progress towards modelling spreadsheet-like data using lists of cells (RQ1): the lists can now be visualised as something that looks and behaves like a table. I have described a method for identifying

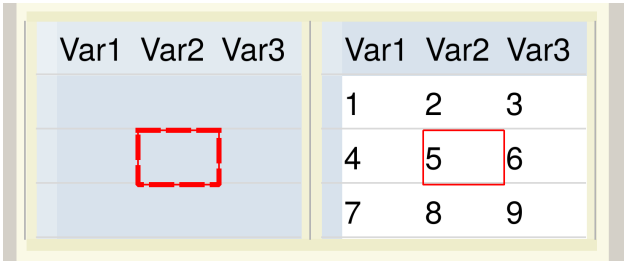
[2]It would be desirable for the editor to provide a reverse visualisation of inheritance as well: given a cell, which other cells does it inherit from? I have not implemented this as yet.



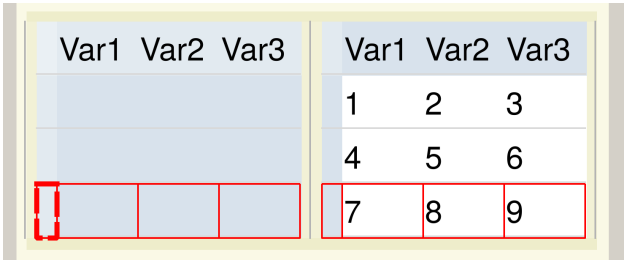
(a) The right hand Var2 column is selected



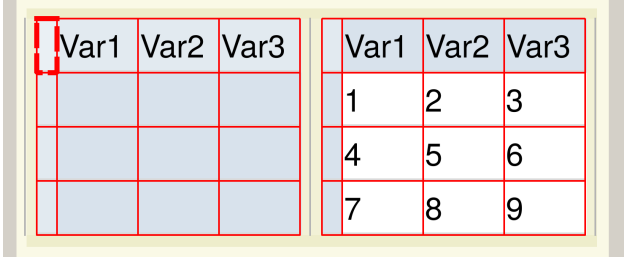
(b) Both Var2 columns are selected



(c) The middle element of each Var2 column is selected



(d) The bottom row is selected



(e) All cells are selected

Figure 5.8: Examples of implicit cell selections

sets of cells that are to be aligned and performing the necessary geometrical calculations for them to be typeset. I then introduced a small prototype editor that allows a user to edit a lish interactively.

I have described a visualisation that addresses the cognitive dimensions of role expressiveness and closeness of mapping. The visualisation method ensures that those lists of cells that have a tabular interpretation are represented as such, with the aid of appropriate shading and gridlines. But it also allows the user to keep sight of the underlying lists, by means of a box view.

Some early implications of the lish for workflow (RQ2) are starting to emerge. In a normal spreadsheet, the user may carry out operations by selecting a range of cells and then invoking some command to manipulate them. The lish equivalent centres around the implicit selections introduced in this chapter. These facilitate the selection of meaningful objects (such as a table column, or a whole table), and hence let the user think in terms of these objects as opposed to arbitrary cell ranges. This style of working will assume further importance in the next chapter, where I will turn to performing numerical calculations.

Chapter 6

Lish calculus

6.1 Review of vectorised arithmetic

This chapter is about doing calculations with lishes. The lish editor provides a spreadsheet-like formula facility where a user may enter a formula in some cell in order to carry out a calculation based on the contents of other cells.

The core of the approach is *vectorisation*: mathematical operations such as addition and multiplication that are defined on numerical arguments are generalised to be applicable to *lists* of such arguments – or more specifically, to lishes. The idea derives from the ordinary mathematical notation for vectors, which allows us to write, for example:

$$2 \times \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 6 \\ 9 \\ 12 \end{pmatrix}$$

In this example, we understand multiplication between a scalar and a vector to mean that each individual element of the vector is to be multiplied by the scalar. And when it comes to addition of two vectors (which must be of the same length), we are to add their individual elements pairwise.

Why should we be interested in vector arithmetic in a spreadsheet-like setting? When Pane and Myers [2006] asked non-programmers to describe programming tasks, they found that "aggregated operations, where a set of objects is acted upon all at once, were used much more often than iteration through the set to act on the objects individually". This insight is the key motivation for vectorisation: we often want to treat a set of cells as a single compound object, rather than iterating through it with a for-loop (in code) or with formula replication (in a spreadsheet). Vectorisation addresses the *closeness of mapping* cognitive dimension, because an opera-

tion that the user would conceptualise as "add these two columns together" is represented directly as such, and not as its decomposition into a series of separate additions on the individual numbers in the columns. Furthermore, the use of just one formula over a multi-cell object results in a DRY representation (see page 30).

Vectorisation is not new to the spreadsheet (subsection 2.3.4). It also has a long history in coding languages, beginning with APL in the 1960s and progressing to modern implementations, notably in data science languages such as Python (with the SciPy library), Julia, MATLAB and R. In this chapter, I shall apply the idea of vectorisation to the lish. I shall build on the existing work in two main ways. First, I shall develop a form of vectorisation that can accommodate the lish as a (possibly irregular) hierarchical structure. Second, I shall introduce a method for simplifying user formulae, by making use of the archetype information that can be obtained from the lish.

The following discussion is framed in terms of the R language, but the others mentioned have similar rules. In R, if we add two numeric vectors of length three, the result is another vector of length three, in which the numbers at corresponding positions in each input are paired and then separately added. R has a *recycling rule* [R Project for Statistical Computing, 2019] which is applied when the vectors being added are of differing lengths: the shorter vector is concatenated with itself as many times as necessary to make it up to the length of the longer before the operation is applied. A common useful case is when one of the vectors is of length one and the other of arbitrary length. The number contained in the shorter vector is in this case added to every number in the longer, as for scalar addition. The rule is extended to allow a vector to be added to a matrix. For example, if a vector of length three is added to a matrix of three rows by four columns, the vector will be added to each column of the matrix. Similar extensions apply to higher dimensional arrays.

Lish calculus builds on these well-established approaches to vectorisation and recycling, but needs to accommodate the fact that a lish may be an irregular structure. Since dimensions in the lish are expressed in terms of depth of nesting, a "depth-wise" version of the recycling rule will be required. This will be introduced in section 6.7, where it will be seen that certain cases can give rise to ambiguity: the recycling could be carried out at more than one level within the structure. Instead of requiring the

user to specify the level explicitly (potentially a *hard mental operation* in cognitive dimension terms), the archetype structure will be used to identify the level. The information captured in the archetypes will also come to the fore in section 6.8, where it will be used to simplify operations involving aggregation.

There is one further way in which lish calculus needs to go beyond R-style vectorisation as I have described it above. A lish (to use a spreadsheet analogy) represents a *whole* worksheet – titles, explanatory notes, scalar constants, independent tables, and all. We do not have a separate type of object for the "workspace" in which the vectors, matrices, etc. to be operated on reside: the lish *is* the workspace. We therefore need a procedure for extracting those subsets of cells that are to participate in any given operation, and a counterpart procedure for writing the results back in afterwards. In the next two sections, I will develop these procedures.

6.2 Extracting data out of a lish

6.2.1 Locating the cells to extract

We shall often want to extract part of a lish (such as a row or column of a table) for input to a calculation. I will use this running example, which contains a small sample of sales data:

$$\left[\begin{array}{c} \bullet, \\ \text{Sales summary}, \\ \left[\begin{array}{lllllll} [\ \text{Region}, & [\ \text{Sales}, & \text{Jan}, & \text{Feb}, & \text{Mar}\ ], & \text{Total}\ ], \\ [\ \text{North}, & [\ \bullet, & 200, & 350, & 300\ ], & \bullet\ ], \\ [\ \text{South}, & [\ \bullet, & 150, & 200, & 250\ ], & \bullet\ ] \end{array}\right] \end{array}\right]$$

Suppose first we want to extract the "North" row. This row is represented as a sublist, so extraction is trivial: we can simply make a copy of the already existing sublist as our extract, namely:

$$[\text{North}, [\bullet, 200, 350, 300], \bullet].$$

But suppose now we want to extract the "Feb" column. This time, the cells of interest do not form a sublist in the original structure – the column "cuts across" the three sublists that form the rows. So we cannot simply copy out an existing sublist to form our extract as was done for the "North" row. Fortunately, the user already has an interactive visual method of identifying the three cells of interest in the form of an *implicit selection*

(subsection 5.4.3). If we adopt the simple solution of creating a new list to hold the extract, we obtain:

$$[\text{Feb}, 350, 200].$$

Next let us extract the "Sales" cell and all its inheritors – that is, the four columns headed "Sales", "Jan", "Feb" and "Mar" respectively. The twelve cells involved once again do not lie neatly in a single sublist. We could try creating a single unstructured new list to contain them, which would yield:

$$[\text{Sales}, \text{Jan}, \text{Feb}, \text{Mar}, \bullet, 200, 350, 300, \bullet, 150, 200, 250].$$

This has all the required content, but some structural information has been lost. It will be important in the context of vectorised arithmetic that the extracted data are treated as an array of three rows by four columns, not as a single vector of twelve elements. So it would seem appropriate when extracting these cells to place them in a list structure resembling the one they were extracted from – but how much of that original structure do we need to keep? A conservative strategy would be to retain the entire sublist structure of the original lish, but filter out all but the twelve cells of interest. We would obtain:

$$\left[\left[\begin{array}{cccc} [\,[\,\text{Sales},\, & \text{Jan},\, & \text{Feb},\, & \text{Mar}\,]\,], \\ [\,[\,\bullet,\, & 200,\, & 350,\, & 300\,]\,], \\ [\,[\,\bullet,\, & 150,\, & 200,\, & 250\,]\,] \end{array}\right]\right]$$

As well as being very inefficient, this strategy obfuscates the true structure of the data extracted. An intermediate form where only "relevant" structure is retained would appear preferable; let us see what that might look like.

6.2.2 Retaining relevant structure

When extracting cells from a lish, I will retain structure according to the following criterion. If *every* element of some sublist in the original lish is either a cell that is part of the extract, or contains at least one such cell somewhere within it, then that sublist is retained in the extract. Otherwise, it is dropped.

Let us apply this criterion to the "Sales" extract above. First, we consider the root of the original lish. One of its elements is the "Sales summary"

cell, which is not part of the extract. So the root is to be dropped from the structure of the extract.

Moving one level in, the main sublist (containing all of the table) is retained. Its three elements are the row sublists beginning "Region", "North" and "South" respectively. Each of these rows does contain at least one cell that is part of the extract (actually, they each contain four such cells – for example, "North" contains *null*, 200, 350 and 300).

Moving in a further level, we consider the sublist beginning "Region", which holds the entire top row of the table. The first element of this sublist is the cell "Region" which is not part of the extract. Therefore this sublist is not retained. Similarly, nor are the sublists for the other two rows.

Finally we come to the innermost level. Each of the four elements of the sublist beginning "Sales" is a single cell that is part of the extract, so this sublist is retained. Likewise, so are the two sublists immediately below it, each beginning with *null*.

Having discarded two of the original four levels of nested lists, we are left with the structurally appropriate result of:

$$\begin{bmatrix} [\ \text{Sales}, & \text{Jan}, & \text{Feb}, & \text{Mar}\ ], \\ [\ \bullet, & 200, & 350, & 300\ ], \\ [\ \bullet, & 150, & 200, & 250\ ] \end{bmatrix}$$

6.2.3 Extracts as traces

Any data extract (beyond a single cell) obtained by the above procedure is clearly a list, but is it a lish? For the sales extract above, the answer is yes. In general, however, once structure extraneous to the extract has been removed, this is not guaranteed. A counterexample would be an extract of the cell containing 5 and all its inheritors in the lish of subsection 4.5.4. This extract is

$$\left[\ \begin{bmatrix}5,\\6\end{bmatrix},\ \left[\begin{bmatrix}7,\\8\end{bmatrix},\ \begin{bmatrix}9,\\10\end{bmatrix},\ \begin{bmatrix}11,\\12\end{bmatrix}\right]\ \right]$$

which is not a lish, because the prior template of the second element is [5, 6] with which this element does not conform.

Fortunately, the vectorised operations to be defined in this chapter do not rely on their operands themselves being lishes. Ordinary lists will do just fine, except for one deficiency. The operations *will* depend on the *archetypes* of the structure from which the data were extracted, so some archetype information needs to be captured. For this purpose I make one

small modification to the extraction procedure above: instead of putting the results in an ordinary list, I shall put them in a *trace* (subsection 4.3.1). The archetype of each trace in this context is simply the archetype of the lish from which it was extracted. For example, suppose in the original Sales lish the archetypes of the root, the outermost sublist, the "Region" sublist and the "Sales" sublists were respectively 0x0040, 0x0080, 0x00c0 and 0x0100. The extract based on the "Sales" cell now becomes:

$$\left(\begin{array}{ccccc} (\ \text{Sales}, & \text{Jan}, & \text{Feb}, & \text{Mar} & \text{0x0100}\ ), \\ (\ \bullet, & 200, & 350, & 300 & \text{0x0100}\ ), \\ (\ \bullet, & 150, & 200, & 250 & \text{0x0100}\ ) \\ & & \text{0x0080} & & \end{array}\right)$$

Earlier, I ascribed informally the list [Feb, 350, 200] to the simpler extract consisting only of the February column. If we follow the more formal procedure above, we obtain the trace (Feb, 350, 200 0x0080) for this extract: the 0x0080 sublist (the outermost one, comprising the whole table) is the only sublist in the original lish for which *every* element is represented by a cell in the extract.

6.3 Writing data into a lish

6.3.1 The 1:1 case

I now consider the converse of the situation in the previous section. Suppose we have completed a calculation, whose result is either a single number or a trace. We now need to write the result back into the lish. For example, in the previous section we might have calculated the row sums of the sales figures, and wish to write the answer (850, 600) back into the "Total" column.

In this example there is a convenient one-to-one correspondence between the cells in the answer and the inheritors of Total, which are currently both *null*. Writing the answer back consists simply of overwriting the two nulls with the two numeric values, and we obtain:

$$\left[\begin{array}{c} \bullet, \\ \text{Sales summary}, \\ \left[\begin{array}{ccccccc} [\ \text{Region}, & [\ \text{Sales}, & \text{Jan}, & \text{Feb}, & \text{Mar}\ ], & \text{Total}\ ] \\ [\ \text{North}, & [\ \bullet, & 200, & 350, & 300\ ], & 850\ ] \\ [\ \text{South}, & [\ \bullet, & 150, & 200, & 250\ ], & 600\ ] \end{array}\right] \end{array}\right]$$

6.3.2 The *n*:1 case

What if we had carried out the same calculation, but wanted the result to be stored separately from the table, instead of in an existing column? If desired, we could create elsewhere a new sublist of three cells (one template-forming cell for the label, and two ordinary cells for the numbers) and write the result there instead. But this would impose upon the user the extra step of allocating the required number of new cells in a standalone list. And there is a more serious problem. The lish is intended to support a spreadsheet-like recalculation model, where the results of formulae are refreshed as necessary upon changes to the data. If the user were to set a formula for the row sums in a standalone sublist but subsequently inserted a row in the table, the standalone sublist (not sharing a common archetype with any part of the table) would not receive a corresponding insertion. The formula result upon re-evaluation would no longer fit.

The lish solves this problem by allowing a multicellular formula result to be written into a single cell – a form of dynamic memory allocation. The receiving cell is promoted to a sublist and the resulting structure can be navigated just like any other sublist. In that sense, it is indistinguishable from content that the user created explicitly. However it is marked internally as being *dynamic* so that the formula evaluator knows to revert it to a single cell prior to recalculation. Hence the new value will be unconstrained by the structure of the previous value. The dynamic marker is also consulted by the graphical rendering procedure (subsection 5.4.1), which draws a dotted border around dynamically generated regions.

6.3.3 The 1:*n* case

The converse of the previous situation occurs when the result of a calculation is a single cell, but the destination is multicellular. This is rather easier to deal with, as no new structure needs to be created. The single value is simply *recycled* over all the destination cells, making a copy in each one (and overwriting any existing content). A typical use case would be to fill an entire column with a uniform value.

6.3.4 The *m*:*n* case

In the most general case, we have m result cells to be written into n destination cells, which might differ both in structure and in extent. One use case is when some dimensions of a structure are static, but others are dynamic.

For example, in the Sales table we might have had a situation where each region was explicitly created by the user, giving a static number of rows, but the monthly data were the result of a formula where the number of months to include was not known in advance. I now give a general procedure for the $m : n$ case.

The sub-cases where either the result or the destination is a single cell are handled as above, so I assume here that the result is in the form of a trace, T. Likewise I construct a second trace, U, formed from the destination cell and all its inheritors according to the procedure of section 6.2. In this context, the trace U is formed by *reference*, so that mutating an element of U is the same as mutating the corresponding element in the receiving lish.

If U is of length one, it is first dynamically expanded to match the length of T, if that is greater than one. For example, if T = (Var1, 1, 2, 3) and U = (Var2) then U is expanded to (Var2, •, •, •) before any further operations are applied.

There are now three sub-cases to consider, depending on the length of T. For all sub-cases, the first *cell* contained within U, which typically contains the label for the result, is specially exempted and is not overwritten. (In the event of a recursive application of the procedure, this exemption is applied only at the outermost level.)

Sub-case 1: T is of length greater than two

We first compare the length of T with the length of U (after the above expansion, if applicable). If the two lengths are equal, each element of T is written into the corresponding element of U, with recursive use of the current procedure. Otherwise, U is filled with an error indicator, "Err!", to signify to the user that the result of a calculation could not be matched to the structure intended to contain it.

For example, if T = (Var1, 1, 2, 3) and U = (Var2, 4, 5, 6) we obtain (Var2, 1, 2, 3). But if U were instead the longer trace (Var2, 4, 5, 6, 7) we would obtain the error-filled result (Var2, Err!, Err!, Err!, Err!). Recall that the first cell in the destination is exempted from overwriting, so Var2 appears in the result in both cases.

Sub-case 2: T is of length equal to two

In this sub-case, the first element of T is written into the first element of U. The sole remaining element of T is recycled over every element of U except

the first. This supports a pattern where a scalar value is represented in a lish of two elements, where the first is a label and the second gives the value.

For example, if T = (pi, 3.14) and $U = (\text{pi\_column}, \bullet, \bullet, \bullet)$ then we obtain $(\text{pi\_column}, 3.14, 3.14, 3.14)$.

Sub-case 3: T is of length one

In this sub-case, the sole element of T is written into every element of U. For example, if $T = (3)$ and $U = (\text{Answer}, 4, 5, (6, 7))$ then on writing T into U we obtain $(\text{Answer}, 3, 3, (3, 3))$. In this example, the $1 : n$ case was applied when it came to writing 3 into the sublist (6, 7).

6.3.5 The *m:n* procedure in action

The procedure above accommodates cases where some of the structure in the result was already present in U, but further structure was contributed by T. For example, with $T = (\bullet, 1, 2, (3, 4, 5), 6)$ and $U = (\text{Var1}, 7, 8, 9, 10)$, we would obtain $(\text{Var1}, 1, 2, (3, 4, 5), 6)$. In this result, the sublist $(3, 4, 5)$ replaced the single cell 9 in the original.

New structure can also appear as a result of a U of length one being expanded. For example, suppose we have:

$$T = \begin{pmatrix} (\ \bullet, & \text{Var1}, & \text{Var2}\ ), \\ (\ \bullet, & 1, & 2\ ), \\ (\ \bullet, & 3, & 4\ ), \\ (\ \bullet, & 5, & 6\ ) \end{pmatrix}, \qquad U = \Big((\ \text{Results}, \quad \bullet, \quad \bullet\ ) \Big)$$

Since T is of length four but U is only of length one (it contains one sublist of three), U is first expanded to length four. In accordance with the lish insertion operation of subsection 4.6.3, each of the three new elements conforms with the prior template established by the first, and we obtain:

$$U = \begin{pmatrix} (\ \text{Results}, & \bullet, & \bullet\ ), \\ (\ \bullet, & \bullet, & \bullet\ ), \\ (\ \bullet, & \bullet, & \bullet\ ), \\ (\ \bullet, & \bullet, & \bullet\ ) \end{pmatrix}$$

There is now a 1:1 correspondence between the cells in T and U, so once

the content of T has been copied over we obtain:

$$U = \begin{pmatrix} (\ \text{Results},\ & \text{Var1},\ & \text{Var2}\ ), \\ (\ \bullet,\ & 1,\ & 2\ ), \\ (\ \bullet,\ & 3,\ & 4\ ), \\ (\ \bullet,\ & 5,\ & 6\ ) \end{pmatrix}$$

The end result is a table that always has three columns, as determined by the original structure of U, but may have any number of rows depending on how many there are in T. If T were to contain more than three columns, U would be flooded with error indicators.

6.4 The lish formula model

In a spreadsheet, the user may set a formula such as "=A1+3" to return three more than the contents of cell A1. They may also define named ranges. For example, if a range named as "sales" is set to refer to cells B3:B17, then the total of the numbers in every cell in that range may be obtained with the formula "=sum(sales)", which is equivalent to "=sum(B3:B17)".

One difference in the lish is that there tend to be far fewer formulae than in an equivalent spreadsheet, due to vectorisation. Just one vectorised formula can generate a multicellular result, which is written across all the cells that inherit from the one containing the formula using the procedure of the previous section. The basic idea, though, is the same as in the spreadsheet: a formula refers (usually) to some other cells, performs calculations on them, and returns a result.

The lish does not have an equivalent of the A1-style coordinate notation for cells; it uses named ranges exclusively, except as noted below. In a lish, however, a named range is not a separate kind of object. The label in any cell with inheritors *is* the name for that cell and all its inheritors. This brings about an improvement on three of the cognitive dimensions:

- Hidden dependencies are reduced, because formulae refer to labels that are present on the "worksheet" as opposed to ranges defined in a separate dictionary.

- Visibility is improved, because the extent of any named range can be seen by the user simply navigating to the naming cell, which will cause the implicit selection encompassing the range to be highlighted immediately.

- Abstraction gradient is reduced, because a "named range" is no longer a separate kind of abstraction and does not need to be defined explicitly. It is simply the range of cells implicitly associated with the given label.

This approach to naming means that the label typed by the user cannot always be used verbatim in the context of a formula that is going to be parsed and evaluated. For example, in a cell labelled "click-counter", the hyphen might well be indistinguishable from a minus sign. To avoid such problems, the editor does some simple sanitisation of names before placing them in a formula – principally, this consists of replacing potentially problematic characters with underscores. Names in formulae are prefixed with a dollar symbol to prevent any ambiguity with function names. For example, sum($sum) means the sum of all the inheritors of a cell labelled "sum".

In a spreadsheet, the user may build a formula interactively by navigating to a cell they wish to include, rather than typing its address. The lish provides a very similar facility. This allows the editor to supply the sanitisation where appropriate and also allows the user to include anonymous ranges – the inheritors of an empty cell. In the latter case, as in Forms/3 [Burnett et al., 2001], the editor supplies a system-generated id in place of the label. The same solution is used in the event of duplicated named cells, except when the duplication is by inheritance, in which case the original cell from which the other ones inherit is deemed to own the name[1].

The formula expression language is implemented as a small internal DSL on Ruby, supporting basic arithmetic and a number of functions including sum, mean and count. Before each formula is passed to the evaluator, a preprocessor resolves any tokens with a dollar prefix to named cells and extracts the data for those cells using the procedure of section 6.2. The data are then operated on (as trace objects, though the user does not need to know that) by the evaluator. The result will be either a single cell or a trace, and is written back into the lish using the procedure of section 6.3.

For simplicity, the editor in its current form does not maintain a dependency graph but simply re-evaluates every formula when asked. It requires that the flow of calculation be in the same order as an iteration over the cells of the lish; formulae may only refer to the results of other formulae that when visited in that order have already been evaluated. Clearly a more efficient approach would be needed if the editor were to be scaled up into a

[1] A facility for qualified names, where named sublists are nested, is work in progress.

practical application.

6.5 Calculations with traces: overall approach

In the sections above I have described how to extract data out of a lish for calculation, how to write the result back in afterwards, and how the user may express these operations. It is now time to turn to how the calculations themselves are performed: to define the rules of vectorisation as they apply to the lish.

I shall be concerned with operations that may be carried out either on cells or traces. For brevity from here on, I shall usually refer to "functions on traces" to encompass functions that can take either type. I shall make the distinction only where the behaviour is different, such as in the terminating case of a recursive procedure.

The overall approach will be to generalise functions that take one or more scalar arguments into equivalent functions that take trace arguments in their stead. It will soon become apparent that certain classes of regular mathematical functions generalise in an equivalent way. For example, once we have generalised the trigonometric function sin, the rules for cos and tan will be immediately apparent. We shall see that a useful way to capture these similarities will be to express them in terms of some higher order functions associated with functional programming. Specifically, in the remainder of this chapter, I shall develop:

- In section 6.6, a function that takes a univariate mathematical function (such as sin, log or sqrt) and a trace; and returns a trace. This is the lish analogue of *map*.

- In section 6.7, a function that takes a bivariate mathematical function, or equivalently (and more commonly) a binary operator such as $+$, $-$, $\times$ or $\div$, and two traces; and returns a trace. This is the lish analogue of *zip*, combined with map.

- In section 6.8, a function that takes a bivariate mathematical function and a trace, and returns a trace of lower dimensionality. This is the lish analogue of *reduce*.

The user is not exposed directly to the higher order functions. For example, they might write a formula "Var1 + Var2" to add two column vectors, and need not know that the machine is translating this to the second

of the three forms above and evaluating "zip_map(Var1, Var2, plus)". It is an approach that supports the cognitive dimension of *consistency*, because once the user understands how (say) addition is to be generalised for the lish, they will readily grasp how other functions of the same form (subtraction, multiplication, etc.) are to be generalised.

At the end of the chapter (section 6.9), I shall consider some other functions on traces that do not fit any of the three forms above.

6.6 Calculations with traces: generalising map

Suppose we would like to apply the square root function, sqrt, to a trace. If every element in the trace is a cell (i.e. there are no nested traces) then the ordinary vector form of map can be applied unchanged. The result is the map of sqrt over the given trace. For example:

$$\text{sqrt}(1, 4, 9, 16, 25) = (1, 2, 3, 4, 5)$$

If, as is common in lish usage, the first element is either a *null* or a label, then this is not an error. A first element that is of a non-numeric type is simply carried over unchanged into the result. For example:

$$\text{sqrt}(\text{Var1}, 256, 81) = (\text{Var1}, 16, 9)$$

There is a simple generalisation to nested traces. All we need to do is apply map recursively. That is, elements of the trace that are single cells are passed just as before to the given function (sqrt, in our example), and elements that are sublists are passed back in to map, along with the same function. For example:

$$\text{sqrt}(\text{Var1}, 1, 4, (\bullet, 9, 16), 25) = (\text{Var1}, 1, 2, (\bullet, 3, 4), 5)$$

6.7 Calculations with traces: generalising zip

6.7.1 Zipping lists of scalars

Suppose we wish to add two traces together. If they are of equal length and all elements are scalars (single cells), then we just have ordinary vector addition. Once again, an initial non-numeric label will be carried over to the result unchanged. For example:

$$(\text{Var1}, 1, 2, 3) + (\bullet, 4, 5, 6) = (\text{Var1}, 5, 7, 9)$$

(If both operands have different labels, the one from the right hand operand will "win" unless it is *null* – the same rule as for composition in subsection 4.3.2.)

Conceptually this may be regarded as a zip followed by a map operation. In the above example, we first zip the two operands to obtain

$$((\text{Var1}, \bullet), (1, 4), (2, 5), (3, 6)).$$

We then pass the zipped form to map along with a function that takes a trace of two elements, and returns the result of adding them together.

6.7.2 Recycling with scalars

Following the same procedure as the R language, we can employ recycling to add a scalar to a trace, or a trace of length one to a longer trace. For example:

$$3 + (\text{Var1}, 4, 5, 6) = (3) + (\text{Var1}, 4, 5, 6) = (\text{Var1}, 7, 8, 9)$$

There is a modified recycling rule when the shorter trace is of length two. In this case, the initial element of the first operand is paired as before with the initial element of the second. The remaining element of the operand of length two is recycled over *every* element of the other operand. For example:

$$(\text{VATrate}, 0.2) \times (\text{price}, 100, 150, 200) = (\text{price}, 20, 30, 40)$$

If both traces are of length three or longer, then their lengths must agree exactly. If they do not, the resulting cells will be filled with an error indicator to alert the user to the mismatch.

6.7.3 Recycling with traces

So far, I have added or multiplied traces where all the elements are cells. A more significant extension to the recycling rule is required when some elements are sublists. Suppose we would like to add the two traces:

$$T = (\bullet, 100, 200), \qquad U = \begin{pmatrix} (\ \bullet, & \bullet, & \bullet\ ), \\ (\ \bullet, & 1, & 2\ ), \\ (\ \bullet, & 3, & 4\ ) \end{pmatrix}$$

Both traces are of length three. So if we follow the procedure that we used for zipping individual cells above, the result will be made up of the following three elements:

1. $\bullet + (\bullet, \bullet, \bullet)$

2. $100 + (\bullet, 1, 2)$

3. $200 + (\bullet, 3, 4)$

The traces in each of the three sums now contain only individual cells, so we are back on familiar ground and the final result is:

$$\begin{pmatrix} (\ \bullet, & \bullet, & \bullet\ ), \\ (\ \bullet, & 101, & 102\ ), \\ (\ \bullet, & 203, & 204\ ) \end{pmatrix}$$

However, there is another interpretation of this sum. Since T is "shallower" than U, we might consider turning the recycling "inside out". That is, instead of recycling each element of T over one element of U, we could recycle the *whole* of T over U. The result would then comprise the following three elements:

1. $(\bullet, 100, 200) + (\bullet, \bullet, \bullet)$

2. $(\bullet, 100, 200) + (\bullet, 1, 2)$

3. $(\bullet, 100, 200) + (\bullet, 3, 4)$

Once again, the problem has been reduced to sums of traces containing only individual cells, but the result now has each element of T added to a column instead of to a row:

$$\begin{pmatrix} (\ \bullet, & \bullet, & \bullet\ ), \\ (\ \bullet, & 101, & 202\ ), \\ (\ \bullet, & 103, & 204\ ) \end{pmatrix}$$

Notice that if we had stayed with the first version of the recycling rule, the sum $(T) + U$, where the parentheses around T denote enclosure inside a further trace, would also have given this result.

Other things equal, I define binary operations on traces as using the first version of the recycling rule, for consistency with the individual cell case. If the second interpretation is wanted, it can readily be obtained by enclosing one of the operands, as has just been seen.[2]

[2] In the formula language of the editor, this enclosure is achieved using a vec() function, since parentheses are reserved for their normal purpose of demarcating subexpressions.

6.7.4 Recycling: mining the archetypes

If this were the whole story, however, the calculation would not always be making full use of the information implicitly supplied by the user when they expressed their data as a lish. Consider the lish below, which shows another version of the earlier sales data.

$$\left[\begin{array}{cccccc} [& \bullet, & [& \bullet, & \bullet, & \bullet, & \bullet &]], \\ [& \begin{bmatrix}\text{Region,}\\ \text{North,}\\ \text{South}\end{bmatrix}, & [& \begin{bmatrix}\text{Sales,}\\ \text{N,}\\ \text{S}\end{bmatrix}, & \begin{bmatrix}\text{Jan,}\\ 200,\\ 150\end{bmatrix}, & \begin{bmatrix}\text{Feb,}\\ 350,\\ 200\end{bmatrix}, & \begin{bmatrix}\text{Mar,}\\ 300,\\ 250\end{bmatrix} &]] \end{array}\right]$$

If we extract a trace based on the Sales cell, we might obtain:

$$\left(\begin{pmatrix}\text{Sales,}\\ \text{N,}\\ \text{S}\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Jan,}\\ 200,\\ 150\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Feb,}\\ 350,\\ 200\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Mar,}\\ 300,\\ 250\\ \text{0x0140}\end{pmatrix} \quad \text{0x0180}\right)$$

In the trace above, I have reverted to including archetype annotations, as they will be important for the next step. It can be seen that all the sublists of the inner dimension have archetype 0x0140, while the single list of the outer dimension has archetype 0x0180.

Ideally, the lish would provide a *range intersection operator* so that the row beginning with "N" in the trace above could be referred to with an expression like "intersect(Sales, North)". As this is not yet available in the editor, I have used the workaround of setting separate labels "N" and "S" on the inner part of the table.

Suppose we would like to express the sales figures indexed relative to the Northern region. Each month, the sales in the Southern region will be shown as a proportion of the sales in the Northern region that same month. We want to divide the Sales trace above by the row for "N", whose trace is

$$(\text{N}, 200, 350, 300 \;\; \text{0x0180}).$$

In this example, the default version of the recycling rule is just what we want: we recycle each individual cell from the divisor with the trace at the corresponding position in the dividend. We obtain:

$$\left(\begin{pmatrix}\text{N,}\\ \text{N,}\\ \text{N}\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Jan,}\\ 1.00,\\ 0.75\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Feb,}\\ 1.00,\\ 0.57\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Mar,}\\ 1.00,\\ 0.83\\ \text{0x0140}\end{pmatrix} \quad \text{0x0180}\right)$$

I note in passing that the current handling of labels (where the label from the right hand operand always takes priority) leaves something to be desired: every row in the above result has acquired the label "N". A concatenation of labels would preserve more information[3]. In practice, it is often preferable for the user to supply their own labels after the event.

Now suppose instead we would like to index relative to the month of January. This time we have to divide the Sales trace by the column for "Jan", whose trace is

$$(\text{Jan}, 200, 150 \;\; 0\text{x}0140).$$

The default recycling rule is neither logically nor mechanically suitable in this case. We would be attempting to pair elements having the Region dimension (expressed by the archetype 0x0180) with elements having the Month dimension (expressed by the archetype 0x0140). And attempting to perform pairwise operations between a trace of length four and a trace of length three will produce an error result.

The appropriate course when undertaking this calculation is to depart from the default recycling behaviour and divide each element of Sales individually by the whole of $(\text{Jan}, 200, 150 \quad 0\text{x}0140)$. Then, each division operation is between two traces of archetype 0x0140. We obtain:

$$\left(\begin{pmatrix} \text{Jan,} \\ \text{N,} \\ \text{S} \\ 0\text{x}0140 \end{pmatrix}, \begin{pmatrix} \text{Jan,} \\ 1.00, \\ 1.00 \\ 0\text{x}0140 \end{pmatrix}, \begin{pmatrix} \text{Jan,} \\ 1.75, \\ 1.33 \\ 0\text{x}0140 \end{pmatrix}, \begin{pmatrix} \text{Jan,} \\ 1.50, \\ 1.67 \\ 0\text{x}0140 \end{pmatrix} \quad 0\text{x}0180 \right)$$

6.7.5 Recycling: a formal procedure

The intuition to be drawn from the previous subsection is that we want to carry out recycling at an appropriate level so that elements having a common archetype are traversed in parallel. If one or both operands have higher dimensions than the two dimensional table of this example, the matching of archetypes might take place at more than one level. I now describe more formally how this matching is to take place.

Let T be a trace of length l, and let U be a trace of length m. We have a function of two scalars $f(x, y)$ which we require to generalise into its inter-trace analogue $F(T, U)$, where F is to be defined so as to pair appropriate

[3]In fact the current version of the editor has yet to implement the propagation of labels described here; it is currently limited to the mathematical parts of the calculations.

cells within T, U and pass them to f. In the division operation, for example, $f(x, y) = x/y$.

If the archetypes of T and U are equal, then $F(T, U)$ is formed by the default recycling rule described above: we simply pair corresponding elements in T, U. Recursive application of the whole procedure may occur if both elements of a pair are themselves traces.

I next define *internal compatibility* as a non-commutative binary relation on traces. T is internally compatible with U if and only if:

- $T[0]$ is a sublist with the same archetype as U; or
- $T[0]$ is a sublist that is internally compatible with U (recursive case).

If T is internally compatible with U then $F(T, U)$ is defined to be a trace having the same archetype as T, and elements:

$$F(T[0], U), \quad F(T[1], U), \quad \ldots, \quad F(T[l-1], U)$$

If T is not internally compatible with U but U is internally compatible with T, then F(T, U) is defined to be a trace having the same archetype as U, and elements:

$$F(T, U[0]), \quad F(T, U[1]), \quad \ldots, \quad F(T, U[m-1])$$

If neither of T and U is internally compatible with the other one, the archetype information is disregarded and we fall back on the default recycling rule, just as for the equal archetype case.

Notice the similarity between this procedure and the composition of two traces (subsection 4.3.4). The difference here is that when the archetypes do not match, we do not know a priori which of the two traces is to be on the "outside" of the recycling. By testing for internal compatibility, we can detect this appropriately before starting to construct the result.

In the Sales examples above, when we indexed to the North region the archetypes of the divisor and the dividend matched, so the default recycling rule sufficed. When we indexed to Jan, we had a case where Sales was internally compatible with Jan, because Sales[0] and Jan both had archetype 0x0140. The result was therefore a trace with elements:

$$F(\text{Sales[0], Jan}), \quad F(\text{Sales[1], Jan}), \quad F(\text{Sales[2], Jan}), \quad F(\text{Sales[3], Jan})$$

Each pair of arguments to F above is a pair of traces of individual cells (no sublists) with a common archetype. So the default procedure now applies, and we simply form pairwise divisions between those cells.

6.7.6 The ternary operator

The ternary operator, as implemented for example in the IF worksheet function in Excel and the ifelse function in R, takes three arguments: a boolean condition, and two values. If the boolean condition is true then the first value is returned. Otherwise, the second one is. For example

```
ifelse(score >= 50, "pass", "fail") # R code
```

given a scalar `score` containing the value 75, would return "pass".

If any of the arguments to ifelse are subexpressions then these are first evaluated, as is conventional for a function – so the expression `score >= 50` would be evaluated to the single boolean value, TRUE, in the example above.

I have generalised the ternary operator to work with traces by repeated application of the zip procedure described in this section. The first two arguments are first zipped into a trace whose "cells" are a wrapper containing two scalars, one picked from each argument following the usual recycling rules. This compound argument is then zipped to the third argument to produce another trace whose "cells" are a wrapper containing three scalars. The resulting object is passed to the trace version of map, along with a function that unwraps each triplet and passes its three elements to the scalar version of ifelse.

6.8 Calculations with traces: generalising reduce

6.8.1 Reducing lists of scalars

The ordinary vector form of the reduce function takes a vector and a binary operator, such as the addition operator. The reduce function initialises a running variable known as the *accumulator* with the first element of the vector. It then iterates over all remaining elements. At each step, the given operator is applied to the accumulator and the current element. The result is written into the accumulator prior to the next iteration. The return value of the reduce function is the final value of the accumulator.

A common application is in forming the sum of a vector. The sum of a vector v may be expressed as reduce$(v, +)$. For example, if $v = [4, 5, 6]$ then the accumulator is initialised to 4. The first iteration produces $4 + 5 = 9$ which is placed in the accumulator. The second and final iteration produces $9 + 6 = 15$, so 15 is the final result.

In a trace where all the elements are cells, this ordinary vector form of reduce can be applied unchanged, with one exception: the first element (typically containing a label) is always ignored. For example, the sum of the trace (Var1, 1, 2, 3) is equal to 6. Even if the first element is numeric, my version of reduce for traces still ignores it, so the sum of (999, 1, 2, 3) is also 6.

If a trace contains a sublist at a position other than the first, it is flattened, again dropping the first element. For example:

$$\text{sum}(\text{Var1}, 1, (\bullet, 2, 3), 4) = \text{sum}(\text{Var1}, 1, 2, 3, 4) = 10$$

A special type of sum is the *piecewise* sum, intended to allow aggregation *inside* an irregular structure that may contain a mixture of scalars and lists. It is defined as follows. Each element of the trace is passed individually to reduce. If the element is a cell, it is first wrapped in a trace containing only a *null* label and that cell; the sum of such a trace is just the original cell, unchanged. A piecewise sum therefore retains any scalar elements and sums any sublist elements. For example:

$$\text{piecewise\_sum}(\text{Var1}, 1, (\bullet, 2, 3), 4) = (\text{Var1}, 1, 5, 4)$$

6.8.2 Reducing in two dimensions

If the first element of a trace is a sublist then we have a two dimensional or higher structure, and the options for how reduce is to be generalised for traces start to multiply. For example, suppose we wish to sum the trace:

$$\begin{pmatrix} (\ \bullet, & \bullet, & \bullet, & \bullet\ ), \\ (\ \bullet, & 1, & 2, & 3\ ), \\ (\ \bullet, & 4, & 5, & 6\ ) \end{pmatrix}$$

We might require:

- the grand sum, 21.
- the row sums (as laid out here), $(\bullet, 6, 15)$.
- the column sums, $(\bullet, 5, 7, 9)$.

The row sums are simply the results of calling reduce$(x, +)$ where x is iterated over each of the inner sublists. The column sums, on the other hand, do not correspond in an obvious way to one or more applications of reduce, since the columns cut across the list structure so they do not

correspond to an actual list upon which reduce might be called. However, the user should not be forced to lay out a table a particular way in order to obtain summation on the desired orientation, and indeed it is common for both row *and* column sums to be required on the same table. Therefore the lish needs to provide both.

For two-dimensional (or higher) data such as the table above, it is useful to think in terms of a "collapsible" dimension over which reduce should be applied. In this example, the rows are associated with an inner level of nesting, and the columns with an outer level. After reduce has been applied, all the lists at the level corresponding to the "collapsible" dimension will have disappeared – they have been "reduced over" – but the structure corresponding to the other levels will remain.

Collapsing on the inner level is straightforward, as we have seen. If we wish instead to collapse on the outer level, the procedure is first to *transpose* the trace. We make the first sublist contain each first element of an original sublist, the second sublist contain each original second element, and so on:

$$\begin{pmatrix} (\ \bullet, & \bullet, & \bullet, &), \\ (\ \bullet, & 1, & 4, &), \\ (\ \bullet, & 2, & 5, &), \\ (\ \bullet, & 3, & 6, &) \end{pmatrix}$$

The dimension we wish to collapse over has now become the inner level, so we proceed exactly as for the inner case and obtain $(\bullet, 5, 7, 9)$.

The grand sum, although it might sound as if it has to do more work, is rather simpler. To form a grand sum, all we need do is visit every non template-forming cell and accumulate the contents. Equivalently, we might collapse on the innermost dimension repeatedly until the result is down to a single cell.

6.8.3 Reducing in three or more dimensions

There is a recursive generalisation of the above procedure, which I have implemented in the editor but will not specify in detail here. Briefly, to collapse upon the outermost dimension of (say) a three-dimensional array, we first transpose at the top level as before so that the outermost and middle dimensions are exchanged; the innermost lists remain intact. Then, we transpose *individually* each element of this newly formed trace. This exchanges the original top level with the innermost level. The level we wished to collapse on has now moved to the inside, so we simply replace

each of the innermost lists in the doubly transposed version of the structure with its reduction. The editor provides a further extension in the form of generalising the "flattening" procedure described for one-dimensional lists above: in a multi-dimensional setting, flattening is carried out to a common sublist level rather than necessarily all the way down to cell level. The distinction becomes necessary when row or column groups are present.

6.8.4 Reducing: mining the archetypes

How is the machine to decide which of the various forms of "sum" the user wants? One way, of course, would be to require the user to specify explicitly. Just as with the binary operators of the previous section, however, this information can in many cases be reliably deduced from the structure. Consider the following lish, containing another variant of the Sales data:

$$\left[\begin{array}{ccccccc}
[& \bullet, & [& \bullet, & \bullet, & \bullet, & \bullet \ ], \ \bullet \ ], \\
\left[\begin{array}{c}\text{Region,}\\ \text{North,}\\ \text{South}\end{array}\right], & & \left[\begin{array}{c}\text{Sales,}\\ \text{N,}\\ \text{S}\end{array}\right], & \left[\begin{array}{c}\text{Jan,}\\ \text{200,}\\ \text{150}\end{array}\right], & \left[\begin{array}{c}\text{Feb,}\\ \text{350,}\\ \text{200}\end{array}\right], & \left[\begin{array}{c}\text{Mar,}\\ \text{300,}\\ \text{250}\end{array}\right] & \left[\begin{array}{c}\text{Total,}\\ \bullet,\\ \bullet\end{array}\right], \\
[& \bullet, & [& \text{Total,} & \bullet, & \bullet, & \bullet \ ], \ \bullet \ ]
\end{array}\right]$$

If once again we extract a trace based on the Sales cell, we might obtain

$$\left(\begin{pmatrix}\text{Sales,}\\ \text{N,}\\ \text{S}\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Jan,}\\ \text{200,}\\ \text{150}\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Feb,}\\ \text{350,}\\ \text{200}\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Mar,}\\ \text{300,}\\ \text{250}\\ \text{0x0140}\end{pmatrix} \quad \text{0x0180}\right)$$

where as before, I have included archetype annotations in the trace. It can be seen that all the sublists of the inner dimension have archetype 0x0140, while the single list of the outer dimension has archetype 0x0180.

The user would like to enter appropriate formulae in the two "Total" cells so that their respective sublists are populated. First let us consider the one towards the top right. The trace extracted from this cell is:

$$(\text{Total}, \bullet, \bullet \ \ \text{0x0140})$$

When forming the sum of Sales, we could decide to collapse on the dimension associated with the 0x0140 dimension, or the 0x0180 dimension, or both. But the trace of the Total column where the result is to be written has archetype 0x0140, which implies that each element in the answer is expected to correspond to a position in the 0x0140 dimension. So we must

not collapse on this dimension; we need it to be retained in the answer. The only alternative is the 0x0180 dimension. Collapsing on that one, we obtain:

$$(\text{Total}, 850, 600 \;\; 0\text{x}0140)$$

which gives the row totals, as required. The formula the user needs to enter in the Total column is simply `sum($sales)`. No further specification is needed as to what kind of sum – that is entirely deducible from the structure.

Now we turn to the other "Total" cell, at the bottom of the table. The formula for this one is `sum($sales)` as well – exactly the same as for the column formula! To see why, we observe that the trace of this cell is:

$$(\text{Total}, \bullet, \bullet, \bullet \;\; 0\text{x}0180)$$

This time the archetype is 0x0180, so it is 0x0140 that must be collapsed upon, and column sums are the result. The general procedure, then, for applying reduce to a trace, is to compare the archetypes present in the trace that is to be reduced and the trace where the results are to be written. Dimensions whose archetypes appear only in the former are to be collapsed, whereas dimensions whose archetypes appear in both are to be retained[4]. If both input and result have the same dimensionality, reduction falls back to piecewise. The result is a notation that is less *diffuse*, in cognitive dimension terms, than one where the level of aggregation is stated explicitly.

With the row and column sums above written back into the original lish, the final result is:

$$\left[\begin{array}{ccccccc}
[& \bullet, & [& \bullet, & \bullet, & \bullet, & \bullet \;\;], \;\; \bullet \;\;], \\
\left[\begin{array}{c}\text{Region,}\\ \text{North,}\\ \text{South}\end{array}\right], & & \left[\begin{array}{c}\text{Sales,}\\ \text{N,}\\ \text{S}\end{array}\right], & \left[\begin{array}{c}\text{Jan,}\\ 200,\\ 150\end{array}\right], & \left[\begin{array}{c}\text{Feb,}\\ 350,\\ 200\end{array}\right], & \left[\begin{array}{c}\text{Mar,}\\ 300,\\ 250\end{array}\right], & \left[\begin{array}{c}\text{Total,}\\ 850,\\ 600\end{array}\right], \\
[& \bullet, & [& \text{Total}, & 350, & 550, & 550 \;\;], \;\; 1450 \;\;]
\end{array}\right]$$

6.8.5 Are there any exceptions?

It is possible for the user to override the above behaviour by supplying an explicit argument to the sum function, to specify at what level (where zero

[4]In the current implementation, each application of sum collapses only the outermost collapsible dimension, so sum needs to be applied twice to obtain the grand total cell at the bottom right. A future version would iterate automatically over all collapsible dimensions.

is the outermost) the structure is to be collapsed. This can be necessary when the result is not being written back into part of the same table as the input but into a separate structure, perhaps involving dynamically allocated cells. For example, the user might place the formula for the sum of Sales in a single cell somewhere outside the main table. If we follow the default procedure above, the single destination cell has no archetypes at all, so both dimensions of the table are interpreted as collapsible and the result is the grand sum. If instead the user wanted column totals, they could use the formula `sum($sales, 1)` to collapse on the inner dimension only.

The necessity for explicit overrides appears relatively unlikely when the input and the result are part of the same table, since the lish idiom is to create the structure so that the dimensions to be collapsed are those present only in the input. The only exception I have encountered arises when the sum forms a sub-expression within a formula. Suppose we wished to express the sales for each region in January as a proportion of the January total. The proportions for North and South are 200/350 and 150/350, or 0.57 and 0.43 respectively. One way the user could compose a suitable formula would be to refer to the cell already calculated with the column total of 350. But if this cell were not present, or the user simply preferred not to split the calculation to use an intermediate result, the user might want to use a formula like `jan / sum($jan)`. This would work when placed outside the table, but not in a column that shared the 0x0140 archetype with the Jan column. In the latter case, the sole dimension present in the Jan column would be interpreted as non-collapsible, so `sum($jan)` would be evaluated piecewise. The user should instead specify the formula as `jan / sum($jan, 0)`. The explicit zero argument states that Jan should be collapsed on its outermost (and only) level. This formula would work both inside and outside the table.

The root cause of the ambiguity is that one would normally store a result that is a single number, such as `sum($jan)`, within a single cell; there is not any obvious way for it to be interpreted as a multi-valued column. It is only when the sum is a sub-expression in the larger formula `jan / sum($jan)` that a repetition of the single value finds any application.

6.9 Future work: other functions on traces

At its most general, a function defined on traces could take any number of traces as arguments, and return any number of traces as its result. A

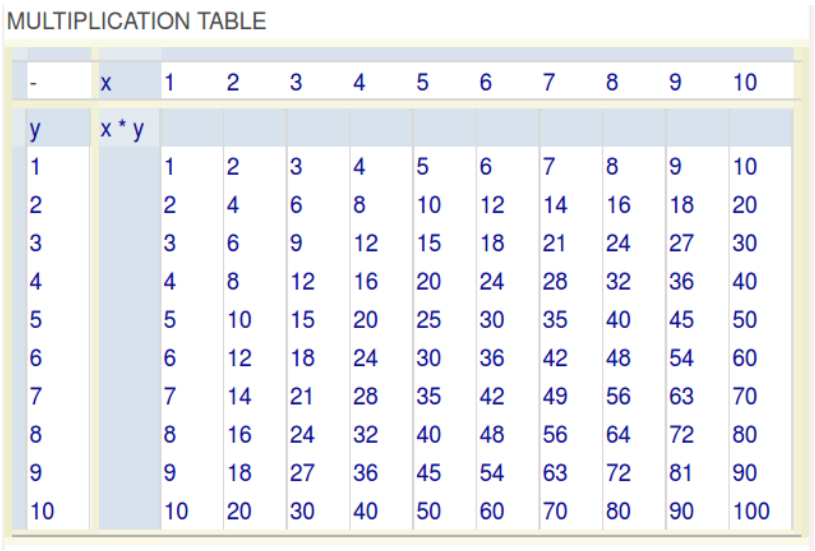
custom function defined in this way would be free to interpret (or ignore) the archetype information as it saw fit. This approach would not be constrained by the rules concerning recycling and collapsing dimensions set out above, and would offer a flexible way of implementing an arbitrary calculation. This flexibility would come at a cost, however: the developer of such a function would have to implement any archetype analysis themself.

The power of using the higher-order functions already defined is that they separate this archetype analysis from the work of the function. For example, once we have the version of reduce from section 6.8 that is applicable to traces, we can easily implement not only the trace version of sum but the trace versions of min and max as well. All we need is a version of (say) max that works on ordinary vectors. We then pass this function to reduce which will take care of all the archetype analysis and partitioning, and assemble the values returned by appropriate applications of max (for example, row maxima) into a final result. In the prototype editor, max and min have been implemented in just this way.

Not all forms of data processing we might want, however, can be expressed in terms of the three higher order functions already defined. So what else is there? In this section, I survey briefly some other functions that would be necessary in a fully operational lish editor. Some of these I have already implemented in "experimental" form, with varying degrees of maturity; typically, these are pragmatic implementations that work for the simple cases of one-dimensional vectors and flat two-dimensional tables, but have not been fully generalised for the lish. For each function I will state the textbook vector form, sketch how it might be generalised for traces, and report on the current implementation status.

6.9.1 The outer product

In the examples examined so far, the product of two vectors has been taken to mean the vector of pairwise products of their elements, with recycling if necessary. In mathematics we have a second type of product, the *outer product*, whose result is a matrix formed from each possible pairing of the elements of the two vectors. An example is shown in the multiplication table in Figure 6.1, where the body of the table is the outer product of the two vectors x and y which form its margins. In R, an outer product (or indeed, an outer version of any binary operation) can be calculated using the `outer` function.



MULTIPLICATION TABLE

-	x	1	2	3	4	5	6	7	8	9	10
y	x * y										
1		1	2	3	4	5	6	7	8	9	10
2		2	4	6	8	10	12	14	16	18	20
3		3	6	9	12	15	18	21	24	27	30
4		4	8	12	16	20	24	28	32	36	40
5		5	10	15	20	25	30	35	40	45	50
6		6	12	18	24	30	36	42	48	54	60
7		7	14	21	28	35	42	49	56	63	70
8		8	16	24	32	40	48	56	64	72	80
9		9	18	27	36	45	54	63	72	81	90
10		10	20	30	40	50	60	70	80	90	100

Figure 6.1: A multiplication table, generated from the outer product of two vectors

I have implemented `outer` in the lish editor, but in its current form it works only on pairs of one-dimensional vectors and is not fully generalised for traces. And there is a further improvement waiting to be made. In the multiplication table example, the table body which is to hold the result contains the archetypes of *both* the input vectors, x and y. Just as we saw when deciding between row sums and column sums earlier, the user's intention in multiplying these two vectors is already encoded in the lish that is destined to receive the result. So a future version should pay regard to those archetypes present in the destination location for binary operations, just as is already done for reduce operations. This would enable the user to type simply `x * y`, and let the machine determine whether the inner or outer product is required. The explicit form would still be needed if the result were to be placed in a standalone table.

6.9.2 The match function

The match function takes two vectors, and looks up the index of each element of the first vector within the second; if the element sought is not present, an index of *null* is returned. For example

$$\text{match}([\text{ham}, \text{eggs}, \text{chalk}, \text{cheese}], [\text{cheese}, \text{eggs}, \text{ham}])$$

gives a result of

$$[2, 1, \bullet, 0].$$

When used in conjunction with the index function (see next subsection), the match function enables lookup tables to be implemented in spreadsheets.

In the lish editor, I have implemented the simple form of match as given above. The following extensions would be desirable to support idiomatic use in the lish.

The simple version searches only for each scalar value within a list of scalars. It would be useful if both the list of items sought and the list in which to seek them were allowed to contain sublists. A match would be defined as the index at which the entire sublist matched, not just an individual scalar. This would enable matching on composite keys.

The simple version returns only the index of the *first* match found, if any. It would be useful if the match for each item sought were allowed to be represented as a list of indices, as opposed to a single scalar. This would capture the presence of any duplicates in the second list, and would form the basis of relational-style table joins.

6.9.3 The index function

The index function takes two vectors, and returns a third vector containing elements picked from the first at those index positions given in the second. For example

index([ham, eggs, chalk, cheese], [3, 0, 2, 2])

gives a result of

[cheese, ham, chalk, chalk].

A special case of index is when the second argument is an *ordering permutation*, that is, when it contains every integer from zero to one less than the length of the first argument, each integer appearing exactly once. A common application is in sorting.

Notice that the index function requires random access to its first argument: it must pick values from a series of arbitrary positions as opposed to processing the elements of this argument sequentially. Unlike with the binary mathematical operators, there is no implicit pairing of elements between the two arguments and no reason to recycle one to the length of the other.

I have implemented the simple form of index above in the editor, and added a small enhancement so that the user may supply a depth argument to discriminate between, say, indexing a table by rows or by columns. A

proper generalisation for the lish would relieve the user of the need to specify this argument where it is deducible from the archetype structure, in just the same way as was done for aggregation earlier. A further generalisation would allow the elements of the indexing argument to be non-scalar; in this case, they would be interpreted as coordinates, allowing elements to be picked from more than one level inside the first argument.

6.9.4 The filter function

The filter function takes two vectors of the same length, where the second consists of boolean values. It returns a vector containing those elements from the first vector where the boolean at the corresponding position in the second vector is true. For example

filter([ham, eggs, chalk, cheese], [true, false, true, false])

gives a result of

[ham, chalk].

I have implemented an experimental filter function in the editor. It employs recycling rules very similar to zip, but since the filtered output is of arbitrary length this output will in general not match the length of either of the inputs. As the length of the output is not known at design time, it is appropriate to use dynamic cell allocation to store it.

The use of recycling allows elements to be included or excluded at the level of whole sublists, as opposed to individual cells. This enables us for instance to filter the rows of a table. For example, given the familiar table

$$\left(\begin{pmatrix}\text{Sales,}\\ \text{N,}\\ \text{S}\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Jan,}\\ \text{200,}\\ \text{150}\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Feb,}\\ \text{350,}\\ \text{200}\\ \text{0x0140}\end{pmatrix}, \begin{pmatrix}\text{Mar,}\\ \text{300,}\\ \text{250}\\ \text{0x0140}\end{pmatrix} \quad \text{0x0180}\right)$$

we might evaluate the expression Jan $>$ 175, obtaining

$$(\bullet, \text{TRUE}, \text{FALSE} \;\; \text{0x0140}),$$

and then use this to filter the table. The presence of the 0x0140 archetype in the trace of booleans causes this trace to be recycled once for each column of the original table, since the columns also have archetype 0x0140. The

result of the filter is then:

$$\left(\begin{pmatrix}\text{Sales},\\ \bullet\\ \bullet\end{pmatrix}, \begin{pmatrix}\text{Jan},\\ 200\\ \bullet\end{pmatrix}, \begin{pmatrix}\text{Feb},\\ 350\\ \bullet\end{pmatrix}, \begin{pmatrix}\text{Mar},\\ 300\\ \bullet\end{pmatrix} \quad \text{0x0180}\right)$$

That is, we obtain only those rows from the original table where the January sales were greater than 175. Note that the 0x0140 archetype has been replaced with *null* at the foot of each column, to avoid violating the invariant that a trace must have length equal to the lish identified by its archetype.

6.9.5 The inject function

The inject function is a more general version of reduce already described, and has yet to be implemented in the editor; the implementation would be similar to the one for reduce. In its simplest form, inject is very like reduce but allows a "seed" scalar to be supplied, which is used in place of the first element of the vector to initialise the accumulator. For example:

$$\text{inject}([1,2,3], 100, +) = 100 + 1 + 2 + 3 = 106$$

A second form of inject retains all the intermediate values of the accumulator (as opposed to only its final value) in the output. With this form, the output has the same length as the input. Example use cases include the cumulative sum and the lag functions (and the lead function, given a facility to traverse the vector in reverse order).

A generalisation specific to the lish would be to allow the accumulator to hold multiple scalars. This would support applications where two coupled columns need to be traversed in parallel.

6.9.6 Towards a general framework

In this section I have considered a diverse selection of functions and the ways in which they might individually be generalised for the lish. This gives rise to a further research question: is there a single unifying framework that would apply to all such functions?

If there were indeed one higher order function to rule them all, it would need to take as its argument a function that works with ordinary vectors and scalars. Its return value would be a function that was in some sense "equivalent" to the input function, but worked with traces. Even this highly intelligent function factory might require some hints in the form of metadata about the function it is being asked to generalise. For example:

- Does it require random access to its vector arguments, or traverse them sequentially?
- Does it produce a result with fewer dimensions than its arguments (like sum), or the same dimensions (like cumulative sum), or higher dimensions (like outer)?
- Does the length of the output match the length of an input (possibly after recycling), or could the length change (as for filter)?

It is not clear whether this ultimate level of generalisation is feasible, at any rate without needing so many options and special cases as to be tantamount to treating all the individual functions separately, as before. There are however some useful steps that could be taken in that direction, because two common operations have been seen to recur across the functional forms that we might be interested in generalising:

1. partitioning a trace into subsets (either single cells, or collections of cells) that are to be processed as a piece; and
2. generating pairs of these partitions, with possible use of recycling.

The immediate future work, therefore, would be to abstract away these partitioning and pairing procedures and use them as building blocks for future lish functions. This would continue to provide flexibility to create functions on a case by case basis, while avoiding the need to retreat to first principles each and every time.

6.10 Conclusion

Calculations with lishes involve extracting relevant portions of the data (as traces) from a containing lish, evaluating arithmetic or other functions on those traces, and writing back the result. In this chapter I have applied an approach based on vectorisation, as found in existing coding languages, and generalised it to accommodate nested dimensions and irregular structures. I chose this approach because it allows users to think about their data in terms of complete objects as opposed to individual cells, avoiding the need for either replicated formulae or explicit iteration over cells.

I used a higher order functions framework, which helped to ensure consistency for related functions. Specifically, I considered in detail the map, zip and reduce functions and their generalisation for the lish. These provide

vectorised forms of univariate functions, arithmetic operators and aggregation functions, respectively. I surveyed briefly a number of other functions, and noted some commonalities in how they might be generalised for lishes; abstracting away these commonalities would be a useful direction for future research.

This approach to calculation addresses several of the cognitive dimensions. Vectorised notation is intrinsically concise, keeping diffuseness low. It improves closeness of mapping, by allowing the user to operate on aggregations of cells that correspond to real-world entities. It also avoids, for the most part, any need for the user to address an object explicitly by its location within the nested structure – a potentially hard mental operation, akin to parsing an expression containing multiple nested brackets. This is further helped by the automated archetype matching, which enables the machine to deduce from context the appropriate level for an operation to be applied. I also showed how the implicit cell selection introduced in the previous chapter builds upon and improves named ranges in the spreadsheet. Abstraction gradient is reduced, because a "named range" is no longer a separate abstraction. By the same token, hidden dependencies are reduced and visibility increased, because the labelled collections of cells are directly visible in the editor.

In the next chapter, I shall apply the lish to some more realistic examples.

Chapter 7

The lish in action

7.1 Introduction

In this chapter, I give some examples of how the lish may be applied to model various forms of data. I begin by considering two "patterns" that often appear as building blocks in spreadsheet applications: namely, structures that span more than two dimensions; and objects that own other objects, represented by spanning column heads. I will show how the lish can achieve greater closeness of mapping than an ordinary spreadsheet with these patterns, and hence better guard against accidental inconsistencies.

I then present a larger (fictitious) example application, the materials testing laboratory. This application shows how the lish can accommodate some of the "messy" features that may occur in real life data. At the centre of this example is a repeating pattern of cells that is not a regular array. The example shows the "cloning" process provided by the editor to handle such structures, and also demonstrates a use case where dynamic cell allocation is required.

These examples also highlight some advantages of the lish formula model. They show examples of DRY formulae, including cases where the object operated on is not merely a simple column. They also show how the capture of hierarchy with the lish can address some problems of relative versus absolute cell references in the spreadsheet.

7.2 A three-dimensional table

In a spreadsheet-like layout, we can represent 3D data in the form of a series of 2D tables, which can be visualised as "planes" to be stacked into an array.

Figure 7.1 shows an example in the form of some cash flow data. There are three 2D tables showing revenue, expenditure and cash flow respectively. The figures in the last table are calculated using a formula, and are equal to the differences between the corresponding values in the first two tables.

It can be seen in the list representation at the bottom of the figure that the first element of the list forming the top level collection of tables is a sublist of four elements, each representing one table row. The columns representing the months have been separated into a column group so that they may be treated as a single object (separate from the department names) for purposes of calculation. The trace of this first table forms the prior template of every subsequent element in the top level list, ensuring that they conform to a common structure that is maintained consistently.

The numbers in the revenue and expenditure tables were entered directly by the user. The numbers in the "Net cash flow" table, on the other hand, are the result of a formula. The formula is located in the "Net cash flow" cell itself. Since all the remaining cells in the bottom table are inheritors of this cell, the result of the formula populates that entire table. The formula is:

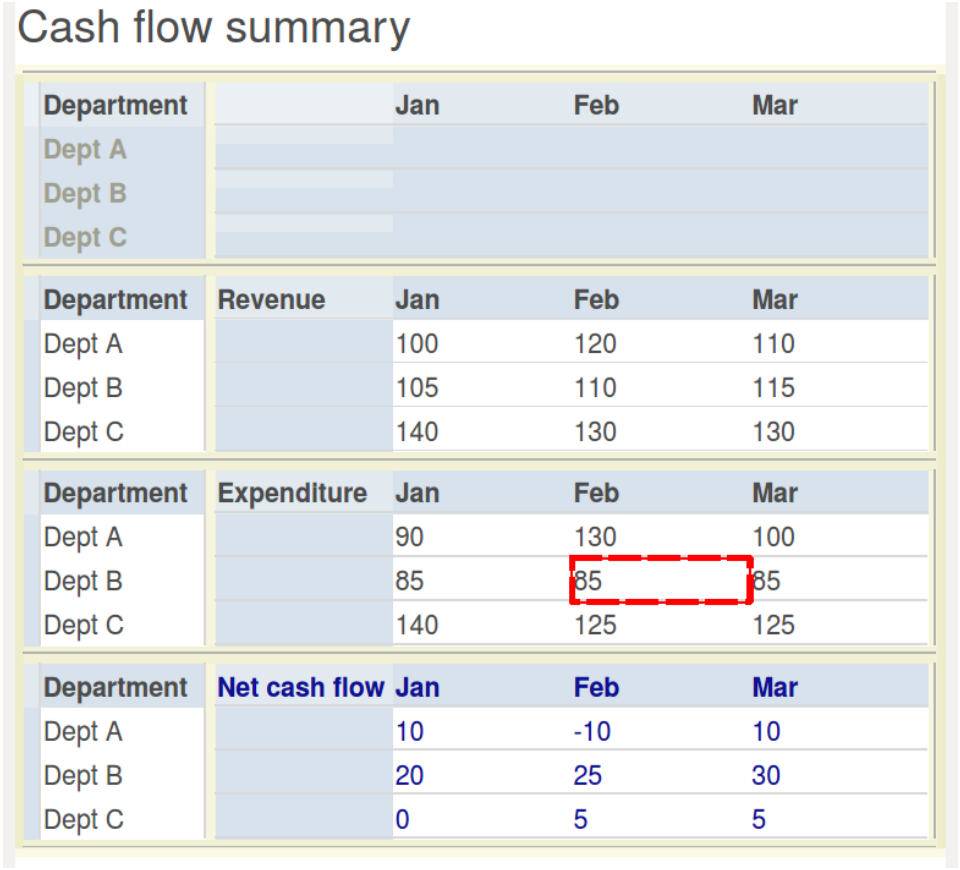
```
$revenue - $expenditure
```

Recall that the dollar symbol denotes a reference to a named cell (it does not have the spreadsheet connotation of an absolute reference). The "revenue" and "expenditure" cells have as their inheritors the top and middle tables respectively, so the result of the formula is a simple matrix subtraction.

Compared to a conventional spreadsheet, the lish representation differs in two main ways:

- There is only one formula, instead of one per cell.
- The structure is inherently three dimensional. If we add a new department, say "Dept D" by inserting a row in any of the individual tables, a corresponding insertion will automatically be made in the others. The scope of the original formula will automatically expand to the extra row as well.

The lish has improved over the spreadsheet on the closeness of mapping cognitive dimension, since the lish reflects the three dimensional nature of the data in a way that the flat grid is unable to do. And this improvement did not come at the expense of increasing the abstraction gradient, because



$$\left[\begin{array}{l}\left[\begin{array}{llllll}\left[\right. & \text{Department}, & \left[\right. \bullet, & \text{Jan}, & \text{Feb}, & \text{Mar} \left.\right] \left.\right], \\ \left[\right. & \text{Dept A}, & \left[\right. \bullet, & \bullet, & \bullet, & \bullet \left.\right] \left.\right], \\ \left[\right. & \text{Dept B}, & \left[\right. \bullet, & \bullet, & \bullet, & \bullet \left.\right] \left.\right], \\ \left[\right. & \text{Dept C}, & \left[\right. \bullet, & \bullet, & \bullet, & \bullet \left.\right] \left.\right] \end{array}\right], \\ \left[\begin{array}{c} \ldots \textit{Revenue data} \ldots \end{array}\right], \\ \left[\begin{array}{c} \ldots \textit{Expenditure data} \ldots \end{array}\right], \\ \left[\begin{array}{c} \ldots \textit{Cash flow data} \ldots \end{array}\right] \end{array}\right]$$

Figure 7.1: A three-dimensional table example, (above) as displayed in the editor, and (below) in skeletal list form

in the lish "everything is a list", so multi-dimensional arrays are not a separate abstraction for the user to learn. (We shall see in the next section, however, that relying on nesting of lists as the sole device for capturing structure can become problematic.) The lish is also a DRY representation, having only one formula where the spreadsheet would need nine.

The representation does suffer from hidden dependencies. If the user navigates, say, to the cell containing zero in the bottom row it will not be obvious to them where this number came from, since the formula producing it is elsewhere (in the "Net cash flow" cell). This is however more of a limitation of the current editor than of the lish itself; it would be very simple to extend the editor to provide this information, perhaps automatically highlighting the cell with the formula. A similar editor limitation affects diffuseness in that the grey template cells take up a large proportion of space in the display. This could be remedied by providing a more concise view in which templates were hidden.

The lish scores quite well on role expressiveness, since the sublist structure and the templates are made visible by the use of shading and grid lines. As was seen earlier in Figure 3.2, the user has the option to visualise the structure more explicitly by turning on the boxes framing individual sublists.

Finally the lish has improved over the spreadsheet on viscosity, because adding extra departments or months causes the whole structure to update automatically, and no new formulae need be created.

7.3 Spanning columns

A common form of data structure occurs when one object "owns" a number of others. We might want to represent the owned objects as a group of rows, or columns, with the name of the owning object as a single label spanning them all. A simple example is shown in Figure 7.2. It shows some revenue data similar to before, but this time the months have been grouped by quarters with the quarter "owning" the three months that sit under it. The Q1 cell, for example, is a single cell spanning all of the Jan, Feb and Mar columns.

In a spreadsheet, this would be accomplished with merged cells. These are only a cosmetic overlay on the grid, so as far as the formula engine is concerned, the cell immediately above all but the leftmost of the columns spanned is the still existing invisible empty cell, not the spanning title (Q1

		Q1			Q2		
Department	Revenue	Jan	Feb	Mar	Apr	May	Jun
Dept A		100	120	110	90	125	120
Dept B		105	110	115	115	110	120
Dept C		140	130	130	145	130	135

$$\left[\left[\bullet,\ \bullet,\ \left[\bullet,\ \text{Q1},\ \text{Q2},\right]\right],\ \left[\left[\left[\bullet,\\ \bullet,\\ \bullet,\\ \bullet\right]\right],\ \left[\begin{array}{l}\text{Dept},\\ \text{DepA},\\ \text{DepB},\\ \text{DepC}\end{array}\right],\ \left[\left[\begin{array}{l}\text{Rev},\\ \bullet,\\ \bullet,\\ \bullet\end{array}\right]\right],\ \left[\begin{array}{llll}[\ \bullet, & \text{Jan}, & \text{Feb}, & \text{Mar}\ ],\\ [\ \bullet, & 100, & 120, & 110\ ],\\ [\ \bullet, & 105, & 110, & 115\ ],\\ [\ \bullet, & 140, & 130, & 130\ ]\end{array}\right],\ \left[\begin{array}{llll}[\ \bullet, & \text{Apr}, & \text{May}, & \text{Jun}\ ],\\ [\ \bullet, & 90, & 125, & 120\ ],\\ [\ \bullet, & 115, & 110, & 120\ ],\\ [\ \bullet, & 145, & 130, & 135\ ]\end{array}\right]\right]\right]$$

Figure 7.2: An example of spanning columns, (above) as displayed in the editor, and (below) in list form

$$\left[\begin{array}{lllll}[\ \bullet, & \bullet, & [\ \bullet, & \text{Q1}, & \text{Q2}\ ]\ ],\\ [\ \bullet, & \text{Dept}, & [\ \text{Rev}, & [\bullet, \text{Jan}, \text{Feb}, \text{Mar}], & [\bullet, \text{Apr}, \text{May}, \text{Jun}]\ ]\ ],\\ [\ \bullet, & \text{DepA}, & [\ \bullet, & [\bullet, 100, 120, 110\ ], & [\bullet,\ 90, 125,\ 120\ ]\ ]\ ],\\ [\ \bullet, & \text{DepB}, & [\ \bullet, & [\bullet, 105, 110, 115\ ], & [\bullet, 115, 110,\ 120\ ]\ ]\ ],\\ [\ \bullet, & \text{DepA}, & [\ \bullet, & [\bullet, 140, 130, 130\ ], & [\bullet, 145, 130,\ 135\ ]\ ]\ ]\end{array}\right]$$

Figure 7.3: A naïve representation of the Figure 7.2 data, in list form. The month names Jan, Feb, etc. will not be treated as column labels with the data organised this way.

in this example). In the lish, spanning cells are part of the underlying structure, and selecting Q1 will select all the columns that it spans. The property of the lish that enables it to express such structures is that a *single* cell in a template may be replaced by a sublist at the same location in a structure that it governs.

Many of the same cognitive dimension considerations apply as in the previous example. In particular, closeness of mapping has again improved in one sense, since the lish has the correct number of cells in the correct places to represent the structure being modelled. However, the lish representation of spanning columns has undermined closeness of mapping in another sense, because it requires the user to play what Green and Petre [1996] describe as a "programming game". Consider the naïve representation in Figure 7.3. On the face of it this looks reasonable enough: it is a legal lish, and appears to have an appropriate structure. The problem is that some cells that we would like to be column headings, such as Jan, Feb and Mar, are not part of the first element of any lish. They are therefore not template cells and do not have the cells in the corresponding columns as inheritors; nor is there any constraint on rows such as DepA to have one cell for each monthly value. The solution is to group the affected rows, as was done in Figure 7.2 – but grouping rows is a different concept to splitting columns, hence the "game". A related problem is that if the user initially entered the data in the naïve form and then upon realising their mistake wanted to group the rows, there is no way of effecting this transformation using only the basic lish operations. The result is high viscosity. The problem might be mitigated by providing higher level refactoring operations to assist with these kinds of scenarios.

As an example of what can be done, Figure 7.4 combines the techniques of spanning titles and multi-dimensional data to model a more complex breakdown of some sales figures. The columns consist of quarters broken down by months as before, and the rows consist of regions broken down by towns. This time a new constraint has been introduced: the fact that a quarter always contains three months has been explicitly captured. This constraint is expressed by the lish containing "Month" and the three narrow cells immediately to its right being of length four. The trace of this lish is the template for the lishes containing $[\bullet, \mathrm{Jan}, \mathrm{Feb}, \mathrm{Mar}]$ and similar. If the user were to navigate to the cell containing "Q2" and execute an insert command, all the new cells needed for "Q3" (including new blank columns

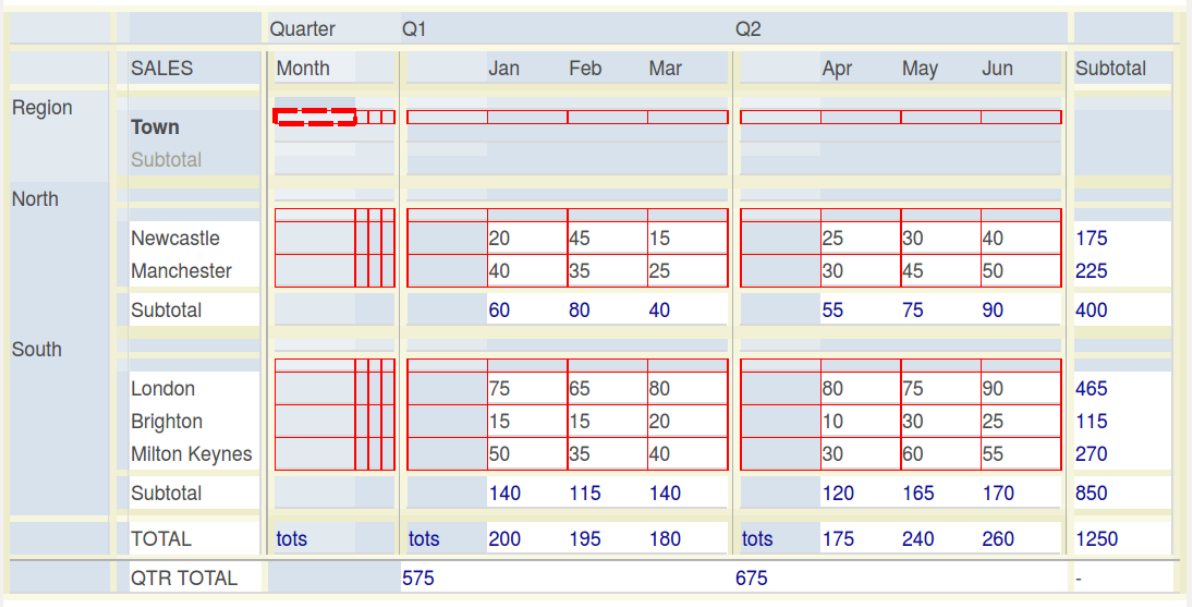


Figure 7.4: An example in which both spanning rows and spanning columns are present

waiting to be populated with July, August and September) would be created in a single operation.

The regions and towns have a similar structure, but with one significant difference: the number of towns per region, unlike the number of months per quarter, is not fixed. This is expressed by the presence of the single cell labelled "Town" which permits a lish of any length to be substituted at those locations where "Town" is the template.

The user has chosen to add individual subtotal rows for each region, and a single subtotal column at the right hand side for all six months. Two further total rows at the bottom give the overall column totals by month and quarter respectively. In principle, only four formulae are required: one for all the column subtotals, one for all the row subtotals, and one for each of the final two summary rows. In the figure, the user has placed the cursor (displayed as the box with the heavier, dashed red line) in an anonymous template cell near the top left, so the editor has highlighted all its inheritors with red boxes. These inheritors comprise all the raw numerical data. Again in principle, all the user need do is define each of the four formulae identically to be the sum of this anonymous cell; the archetypes present in the input range and the four different output ranges in each case give sufficient information to deduce which dimensions (town, month, etc.) need to be summed over.

In practice, the editor in its current state does not yet fully implement the archetype mining of subsection 6.8.4[1], so it needed a little help in this example: some of the formulae were written using the method of subsection 6.8.5 employing an explicit level parameter, and the row subtotals are actually defined per region, rather than the ideal of a single formula for the entire column. A future version of the editor would implement the archetype mining procedure completely and would not need these workarounds.

Implementation issues aside, this example may be pushing the limits of the lish in other ways. The diffuseness that was seen earlier in the 3D example has grown again, to the extent that there are now more template cells than ordinary numerical values, so the available screen space is not used at all efficiently. And the "programming game" noted with the simpler spanning headings in the previous example has now become really rather complex. Figure 7.4 in fact involves no fewer than *nine* levels of nesting, which I do not show in list form here. These arise because new levels are required not only for the four obvious dimensions of Region, Town, Quarter and Month but also to create the spanning titles and to segregate the various levels of subtotal from the raw data. A dedicated tool or "wizard" for creating common structures like this one might be easier for the user than having to play the programming game. Meanwhile the editor does provide some help in the form of visual aids: the shading and coloured boxes go at least some way to ease navigation of such a complex structure.

7.4 Example application: materials testing laboratory

7.4.1 The scenario

This section is based on the following fictitious scenario. A materials testing laboratory routinely makes measurements of the density (mass per unit volume) of samples of material. The mass and the volume are measured separately and then combined. The mass of each sample is determined by weighing it on an electronic balance. The volume is determined by image analysis, in an automatic apparatus which scans the sample from multiple angles. However the volume measurement is less accurate, so it is repeated

[1] In particular, it takes a shortcut by examining only the archetypes in the *first* element of the result. This is sufficient for regular arrays, but can break down for more complex structures like this one.

four times per sample and the mean reading is taken. In addition, a calibration sample of known volume is measured at the start of each session. This is used to derive a calibration factor for the apparatus that must be applied to all subsequent samples that session.

7.4.2 Building the structure

Figure 7.5 shows a lish application to perform the calculations for this measurement. The user records a single calibration factor for the session, and then for each sample they record a sample code, one mass reading, and four volume readings taken directly off the instrument. Each of the volume readings has to be corrected by multiplying it by the calibration factor. Then, the density is calculated as mass divided by volume. This gives four density readings; the mean is taken for each sample.

The core repeating block of the application is the pattern of cells representing a single sample, comprising the sample code, mass, and the table of volumes and densities. Each such sample block is a single sublist, with further structure inside to create the two initial fields and the table. All of these blocks reside in a single enclosing list. The first block in this list (near the top of the figure, with all the cells in grey and no readings populated) represents not an actual sample but a template, that ensures all subsequent sample blocks have the same structure. The user may insert a new, blank sample block simply by selecting the last existing one and invoking the insert operation.

This structure was created by starting with the block for the first sample only – an example of "start constant, then vary" [Victor, 2012]. To support this way of working, the editor provides a shortcut "clone" procedure that takes the user's currently selected lish and combines the following operations:

- Enclose the selected lish by one level.
- Make a copy of the selected lish within the new enclosure, resulting in a pair of lishes having the same structure as the original.
- Move all the original formulae and labels into the *first* lish of the pair.
- Move all the original literal values into the *second* lish of the pair.

The intention is that the user should be able to go from the particular to the abstract, not the other way round. The clone procedure helps the user to

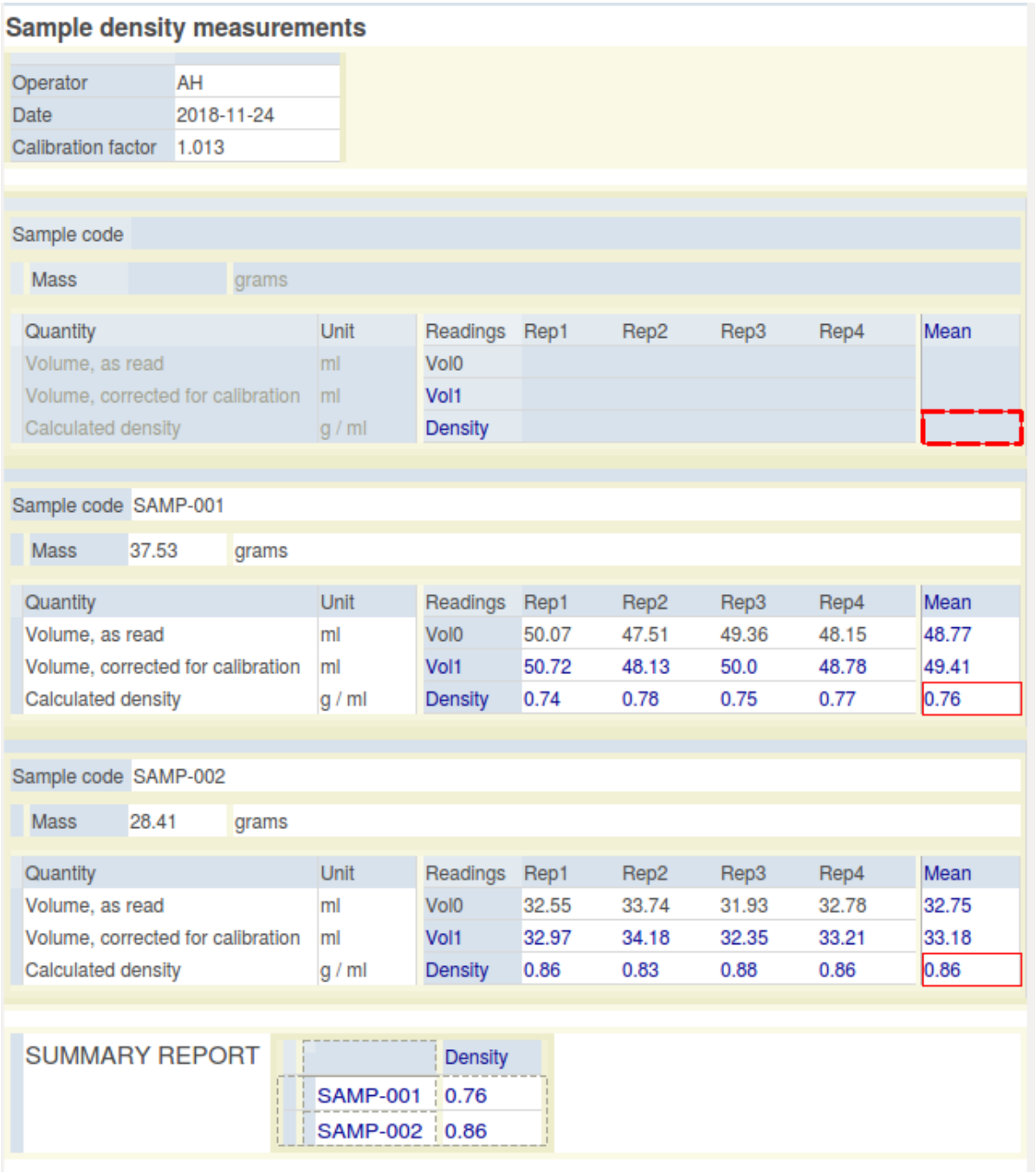


Figure 7.5: The materials testing lab application in the lish editor

separate properties that belong to samples in general from properties that belong to SAMP-001 in particular. But the user can make a start with a concrete first sample *before* having to think about the distinction.

This style of working involves low premature commitment, since the user can develop the template but change it after it has been cloned, and the clones will be updated as well. The effect is similar to copy-paste tracking [Hermans and Van Der Storm, 2015] but applies to any new formulae that might be added to the template (and to new structure, as well) – not just to those formulae already tracked. In common with a normal spreadsheet, the editor also provides progressive evaluation: the result of each formula can be seen as soon as it is created.

7.4.3 Implementing the formulae

The template sample block contains three formulae. First, the Vol1 cell contains:

```
$vol0 * $calibration_factor
```

The Vol0 cell has as its inheritors not only the cells in the four Rep columns on its immediate row, but also all the equivalent Vol0 rows in all the other sample blocks. The calibration factor label (near the top of the figure), on the other hand, has just one inheriting cell: the scalar 1.013. Following the recycling case of subsection 6.7.2, the 1.013 will be multiplied separately by each of the Vol0 cells.

Next we have the Density cell. Its formula is:

```
$mass / $vol1
```

This formula refers to two cells, Mass and Vol1. The trace extracted from the first of these is:

$$\begin{pmatrix} \begin{pmatrix} \text{Mass}, & \bullet \end{pmatrix}, \\ \begin{pmatrix} \text{Mass}, & 37.53 \end{pmatrix}, \\ \begin{pmatrix} \text{Mass}, & 28.41 \end{pmatrix} \end{pmatrix}$$

while the trace extracted from the second is:

$$\begin{pmatrix} \begin{pmatrix} \text{Vol1}, & \bullet, & \bullet, & \bullet, & \bullet \end{pmatrix}, \\ \begin{pmatrix} \text{Vol1}, & 50.72, & 48.13, & 50.00, & 48.78 \end{pmatrix}, \\ \begin{pmatrix} \text{Vol1}, & 32.97, & 34.18, & 32.35, & 33.21 \end{pmatrix} \end{pmatrix}.$$

The outer lists of these extracted traces share the same archetype, namely the original list in which all the sample blocks reside. The elements of the traces are therefore paired one-to-one for the division operation. In each case, this entails dividing a list of two by a list of four. The *null* elements all yield *null* results. When we come, say, to

$$(\text{Mass},\ 37.53) \div (\text{Vol1},\ 50.72,\ 48.13,\ 50.00,\ 48.78),$$

the procedure of subsection 6.7.2 again applies, as was the case for the calibration factor calculation above. The 37.53 from the Mass trace is recycled over each of the four numerical values in the Vol1 trace, so that separate division operations are carried out between the single mass reading and each of the four volume readings.

If a similar application were being developed in a spreadsheet, the developer would typically create the formulae for the above calculations by composing just one formula initially, and then copy-pasting it to other relevant cells. This would require care with the use of relative versus absolute cell referencing (the latter, in an ordinary spreadsheet formula, denoted by dollar prefixes on the row or column coordinates, or both). References to the calibration factor would be absolute, since a single value applies to the whole sheet. References to individual volume readings would be relative, since these references must be updated both by reading number and sample as the relevant formulae are copied. References to mass would be row-relative but column-absolute, since one mass applies to all four volume readings lying on a common row, but there is a separate mass for each sample, on a separate row.

Getting the dollar signs right in a formula is potentially a hard mental operation for a spreadsheet user. Furthermore, in some instances where there is a mix of local and global referencing, there is no combination of relative and absolute referencing that will allow formulae to be safely copy-pasted; the current example would suffer from that problem if the four volume readings for each sample were arranged vertically instead of horizontally. The lish avoids the distinction between relative and absolute references altogether, because the hierarchical structure already distinguishes between those values that are global and those that are local to some object. The recycling rules above ensured that the calibration factor used was always the global value, the mass used was always the sample-local value, and the volume used was always the reading-local value.

The third formula is located in the Mean cell. This formula is:

```
mean($readings)
```

The trace extracted from the Readings cell is a list of three tables, one from each sample block. Each of these tables contains the four rows (Vol0, Vol1 and density, plus the column headings) and five columns (Rep1, Rep2, Rep3 and Rep4, plus the row headings) relating to that sample. So the whole trace has three levels of nesting: replicate, within measurement type, within sample. The Mean column that is to hold the result has a trace containing the last two of these levels, but not the first. By comparing the archetypes in these two traces (as in subsection 6.8.4), the machine has deduced that means need to be taken over the replicate level. Hence individual row means are obtained with no explicit direction needed from the user.

Two further formulae are used in the summary report at the bottom of the figure. These are simple formula-copies of the sample codes and mean densities, respectively. The dashed red cell cursor in the figure is located on the (anonymous) cell that is referenced in order to obtain the latter. These formulae generate dynamically allocated lists of cells, so if samples are added or deleted the summary report will expand or contract accordingly.

7.4.4 Further remarks

In this example, there is an excellent closeness of mapping between the lish and the problem domain. The structure of each sample, with fields for sample code and mass followed by a table for the readings (with column grouping within it), is faithfully captured. The template behaviour of the lish is a natural fit for this repeating structure in which each sample has the same form. The lish also allows selections of non-adjacent cells that represent a single object, such as "all the sample masses", to be treated as such. At the same time, it is sufficiently flexible that properties of an object that are not logically shared with its neighbours are not imposed upon those neighbours. For example, the six cells at the top including the calibration factor have different widths to those immediately under them in the main table. This is as we would wish, since alignment of those cells would not represent any concept that is meaningful to the application – but a normal spreadsheet grid would nevertheless enforce it.

Most of the formatting in Figure 7.5 is the default supplied by the editor. A small amount of secondary notation has been applied manually: the main title and summary report title are in a larger font, and some column widths have been adjusted. The shading and gridlines were all generated

automatically from the lish structure as described in subsection 5.4.1. The generated formatting does a reasonable job of visualising the structure, so there is less need for users to supply these visual elements themselves, as they might do in a spreadsheet.

As with the cashflow example, hidden dependencies are present; the cost of very DRY formulae is that in the density calculations, the formulae are rather distant from the cells that they affect. Once again, it would be a straightforward improvement to the visualisation to make these dependencies more explicit. In designing the editor, I adhered to the spreadsheet convention that the data are visible but the formulae invisible until a formula cell is selected. The lish has so few formulae, however, that it might be preferable to display both data *and* formulae by default; the penalty in screen space would be negligible compared to doing the same thing in a spreadsheet.

7.5 Conclusion

In this chapter I examined lish representations of a number of practically useful structures: 3D tables, spanning headings, and a more complex example having mixed scalar and tabular data with repeating patterns.

The main advantages from using a lish over a spreadsheet for these structures arise from the improved closeness of mapping in the lish. This allows the machine to automate the generation of repeating patterns and to ensure consistency as data are added or deleted. Formulae are simplified, in particular because the way in which lish calculus is defined over hierarchical structures obviates the need for explicit absolute versus relative references. The editor promotes a workflow where the user can start from a small concrete example and subsequently add more data, or even more dimensions, without having to change the original formulae. Hence premature commitment is avoided.

Some limitations of the lish were also apparent. The presence of hidden dependencies and a tendency towards diffuseness might be problematic, although these could be addressed in a straightforward way by an improved editor. Of greater concern was the need for "programming games", especially in the representation of spanning headings. The picture on viscosity is mixed. The ability for the user to revise a formula in a template, or to add a row for more data, and in both cases have the changes automatically reflected elsewhere in the application, certainly reduces viscosity. On the

other hand, the lack of support (currently) for more fundamental refactoring of lish data rather increases it.

Chapter 8

User study

8.1 Purpose and scope of study

Among my original research questions, I asked:

> What would be the consequences for analytical workflow of using a lish, instead of a grid, as the basis of a spreadsheet-like environment?

and

> Where would this alternative representation be located in the space of the cognitive dimensions?

Until now, I have approached these questions as a desk-based exercise. This has yielded partial answers. For example, in the previous chapter I demonstrated some example workflows with the lish, and commented on how these differed from building similar models with the spreadsheet. On the cognitive dimensions side, I have also made some progress simply by inspecting worked examples. In this chapter, I seek a fuller answer by involving some actual users. To assess workflow, I observed those users at work with the lish; to assess the cognitive dimensions, I interviewed them to find out how they experienced and understood it.

The focus of the study was on expert analysts rather than casual spreadsheet users, as the lish (in its current form, at least) is not really well suited to the latter. The participants were recruited from the UK Government Operational Research Service (GORS) and related analytical professions. GORS covers a diverse range of methodologies including forecasting, optimisation, system dynamics, queue modelling and statistics. All participants

were power spreadsheet users, and in most cases had coding experience as well.

Confining the study to a single professional group clearly limits the extent to which the results might be generalised to other spreadsheet users (accountants, say). Resource constraints precluded repeating the study elsewhere, so we should be cautious in generalising beyond the study population. This limitation is however mitigated a little by the diverse range of applications and model structures encountered within GORS.

8.2 Study design

The study was carried out to a design approved by the university's Human Research Ethics Committee under reference HREC/3016/Hall. The interviewer's script is appended at Appendix A.

8.2.1 Format of the study

Twelve volunteers were recruited from GORS and related analytical professions. In the results below, they are referred to anonymously by randomly assigned letters of the alphabet. The volunteers first attended a group presentation, in which I summarised the background to the lish and provided a demonstration of the lish editor. I then conducted a one-to-one session with each participant. This consisted of a warm-up task for familiarisation with the editor, a main task which the participant was asked to carry out unassisted so far as possible, and a semi-structured interview.

8.2.2 Task description

Participants were provided with a "crib sheet" of editor functions, commands and shortcut keys, shown at Appendix B. They were given the small spreadsheet shown in Figure 8.1, showing weekly production figures for a gizmo manufacturer, and were first asked to replicate it in lish form (with the total calculated using a formula). They were then supplied with data for two further sites and asked to extend their lish to encompass those sites as well. Finally they were asked to write a formula that would list the managers of sites whose weekly production exceeded 100 units. A model solution in lish form is shown in Figure 8.2; the frames have been turned on in the editor to show a representative portion of the nested structure.

Site	Arkwright					
Manager	John Smith					
Output	Mon	Tue	Wed	Thu	Fri	Total
week1	23	22	24	20	18	107

Figure 8.1: The initial spreadsheet for the task in the user study

Of course, the task had to be kept very small in order to fit within the time constraints of the session and to be suitable for a user who had only just met the lish. But it did exercise the fundamentals of lish modelling: using lists to make tables, and applying formulae upon ranges of cells via a single template cell. It also exercised some concepts with no direct counterpart in spreadsheet modelling, namely:

1. Column groups. Although spreadsheet users may mark off groups of columns using secondary notation such as vertical lines, they are not an intrinsic part of the structure. In a lish, columns that are to be operated on as a single object must be grouped.

2. The use of templates to initiate a family of similar objects. A template based on the 'Arkwright' data was used in this task to obtain a consistent structure for the other two sites.

3. Ranges consisting of non-adjacent cells. Such ranges are available in spreadsheets if the user selects each cell individually, but in a lish they can be selected via a single template cell.

4. Dynamic allocation. The final query was of a form that might produce an output of arbitrary size, but which could be embedded in a single cell.

8.2.3 The interview questions

An initial question asked about the participant's impressions of an example dataset in long compared to wide form.

The next set of questions concerned the task. Their aim (along with observations on how the participant carried out the task) was to assess the lish in a cognitive dimensions framework. There is a difficulty here: the user interaction is with the actual editor presented to them, so we cannot

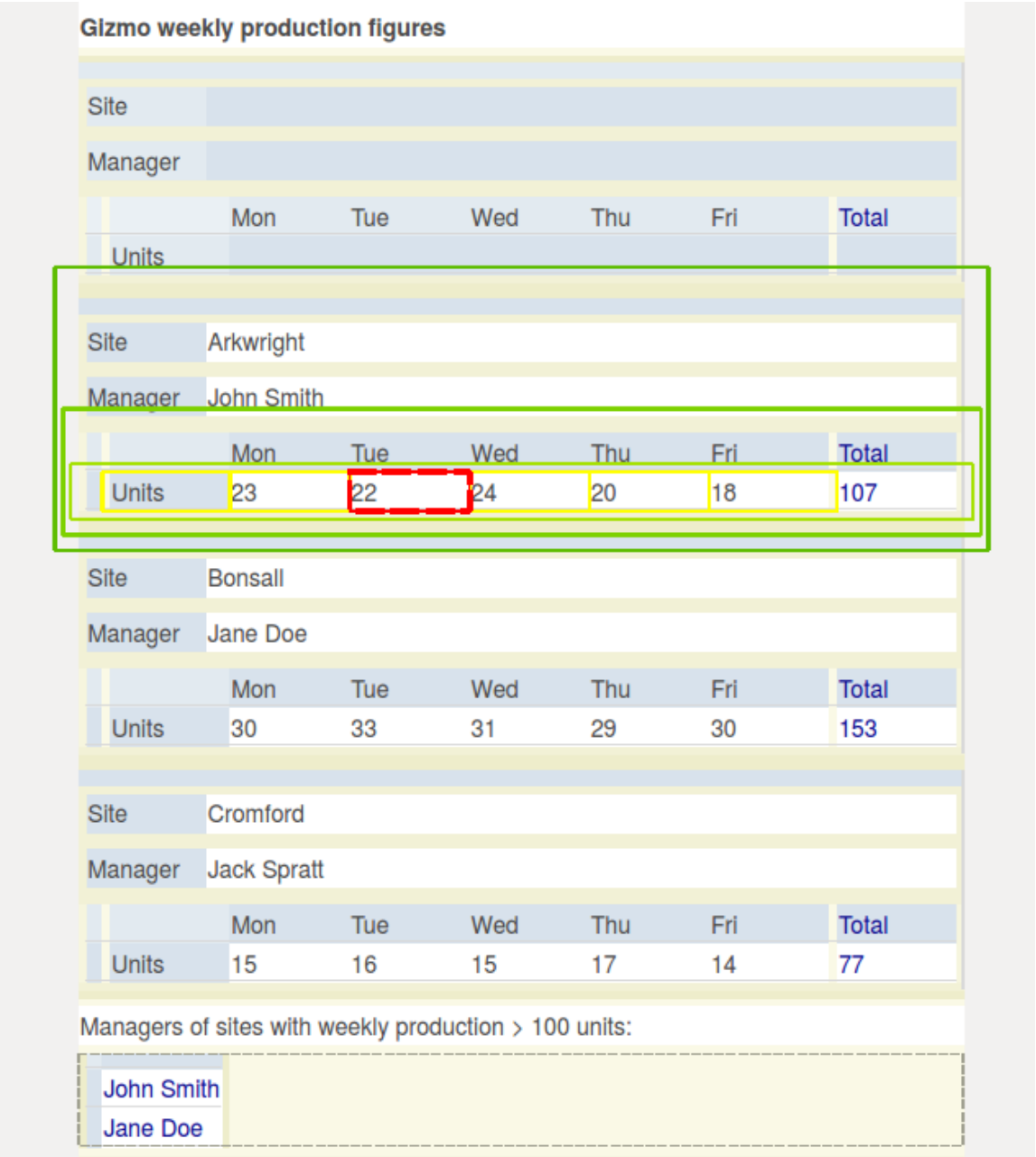


Figure 8.2: A model solution to the task in lish form, with the extra sites added

entirely separate to what extent their experience is driven by the editor itself (which is a rough prototype having no great pretensions to facility of use) and to what extent by the underlying formalism (which is hypothesised to be well suited to the task, and is what we would like to evaluate). The "notation" whose cognitive dimensions are being assessed in this trial can only be the lish as visualised on screen, along with the editor that forms its environment, rough edges and all.

This problem cannot entirely be resolved until a better editor is available! However, as noted in subsection 2.4.1, certain dimensions are heavily driven by the underlying representation, even though in the final analysis they can only be judged in the system as a whole. So as long as the editor is *good enough*, we might still expect to obtain some relevant information on how the lish as a model influences those dimensions. The ones I focused on in this study were:

- closeness of mapping
- abstraction gradient
- role expressiveness
- viscosity.

Participants were asked in this section to reflect on how easy or otherwise they found various aspects of building the model, from the point of view both of forming the structure and implementing the calculations. There followed two more open questions about whether the behaviour of the system was surprising or its operation felt unwieldy.

The other main sequence of questions looked at the relationship between the lish and the participant's own work. The point of these questions was to take the RQ on effect on workflow from a different angle. It has been fairly easy to show that the lish can have an effect (and a beneficial one) on *some* workflows, but do those actually reflect the practices of real world analysis? And if so, do users perceive the aspects that have changed as being of any importance?

Finally, the participant was given the opportunity to make open comments about anything not already covered.

8.3 Results

8.3.1 Wide vs. long

As expected, a strong majority of participants (11/12) found the wide form of the data more readable to the human eye. The sole dissenting participant was someone who carried out a lot of database work, so maybe was so accustomed to handling data in long form that they found the relative unfamiliarity of the alternative a barrier. There was also a majority (10/12) in favour of the long form as an import format for code-based tools. The two who disagreed here were less expert in using such tools, so may simply have been unaware of the issues that "untidy" data would create.

8.3.2 The task

All the participants required at least some hints in order to complete the task. Five of them completed it with some minor hints, and another four with quite detailed hints. By "hints" I mean giving some form of verbal assistance that stopped short of actually telling the participant what to do. Examples were providing reassurance, drawing attention to helpful sections in the crib sheet, and reminding them of something they had seen during the warm-up task. I also gave some commentary on what had happened if the result appeared puzzling, and helped with correcting missteps (since the lish editor currently lacks that indispensable real-world feature, an undo facility). The remaining three participants found the task harder, and I gave them explicit directions in several places.

There was one part of the task for which I had intentionally not included a direct parallel in any of the introductory material, in order to see whether participants would think "in a lish way" about something that they had not actually seen before. This was the formula for the sum of the daily production figures (Monday to Friday). In a spreadsheet, one would select a range comprising the relevant five cells to form the argument to the sum function. In a lish it is the same five cells that are required, but they must be in a sublist before it is possible to select them. The way to do this is to put them in a column group within the table. How to form a column group had been covered in the introduction, but its application deliberately left vague to see whether participants would work it out for themselves.

Only three of the twelve participants deduced unprompted the need for these columns to be grouped during the task. This could be seen either as

a poor reflection on the lish as a representation, or an impressive reflection on the resourcefulness of the three who succeeded (sounds of satisfaction, or even triumph, were audible when the penny dropped). Either way, it is clear that this form of lish usage is not immediately obvious to a new user.

A related issue was that eight of the participants, at their first attempt, created a sum formula that contained a circular reference: in the absence of a column group, the cell that was intended to contain the sum was included in the range to be summed. The number might have been higher still if I had not intervened with some participants who were making slow progress to head them out of this blind alley. The workaround used by most was to place the cell with the sum outside the table entirely. This gave the correct answer even though the layout differed from the more idiomatic solution.

Another common difficulty arose from enclosing material to too great a depth: seven participants at some point within the task were observed to make this error. The participants sometimes seemed to prefer creating extra levels when all that was required was to compose items sequentially. Being at the wrong level in the structure to perform the desired operation was also experienced by five participants. For example, to pull adjacent items into a list, the cursor needs to be at the level of the list itself rather than inside it. These errors may have had more to do with the rather rudimentary interface of the prototype editor as opposed to any misconception about the lish structure desired.

In a few instances, the user selected the first ordinary cell in a lish, rather than its template, when wanting to operate upon a whole lish. This may have been learned behaviour carried over from the spreadsheet, where typically one might enter a formula in the first cell of a column, and then fill down the column. The lish interface could perhaps be made more direct by supporting this mode of interaction: the user would select an example instance of the required data, rather than a more abstract entity in the form of the template, and the machine would generalise appropriately.

Two users, interestingly, constructed independent parallel lists for the weekday labels and the production figures, instead of combining these into a true table. This was a perfectly legal solution though would not have been as robust as the model solution to future change: for example, if the sites started to work a six day week and Saturday was added to the first list, a corresponding cell would need to be added manually to the second list. These users may have been visualising the data as parallel lists rather than

a single table object.

As noted earlier (page 85), the editor renders empty template cells as compressed, in order to reduce the amount of screen space wasted in empty grey areas. One participant mentioned specifically that they found this confusing. The compressed cells do not stand out very clearly when the user is seeking to form their associated selection, and might make it harder visually to parse the structure. On the other hand, a different participant later mentioned that the screen space occupied by all the template cells could be a problem with a large model. An improved interface might offer both a development view where these cells are rendered at full size, and a final typeset view in which they are compressed, or even hidden altogether.

Almost all participants made some comment along the lines that they had some difficulties with the task due to lack of familiarity with the system, but felt that they would find it much easier with a little more practice. For example:

> "[It was] more down to my familiarity with the structure rather than the model itself. I think once I became more familiar I would find it a lot better." – *Participant V*

> "I think if I did it again I'd be better at it!" – *Participant W*

> "I'm finding it difficult to get my head round it now but I can tell as soon as [I have] it will be fantastic, it's just the initial getting your head round it because it works slightly differently to what you're used to." – *Participant N*

These were spontaneous comments as the questionnaire did not ask specifically about this aspect. They are somewhat reassuring as they suggest that many of the difficulties observed were simply due to lack of experience rather than conceptual misunderstandings of the lish.

8.3.3 How easy was it to build the structure?

All but one of the participants at least tended to agree with the statement, "I could easily see how to populate cells and lists" (Figure 8.3). A majority (though less strong) agreed with, "I understood the 'template' behaviour of the first item in my lists, and what patterns were going to repeat as a result" (Figure 8.4). These questions were intended to test abstraction gradient,

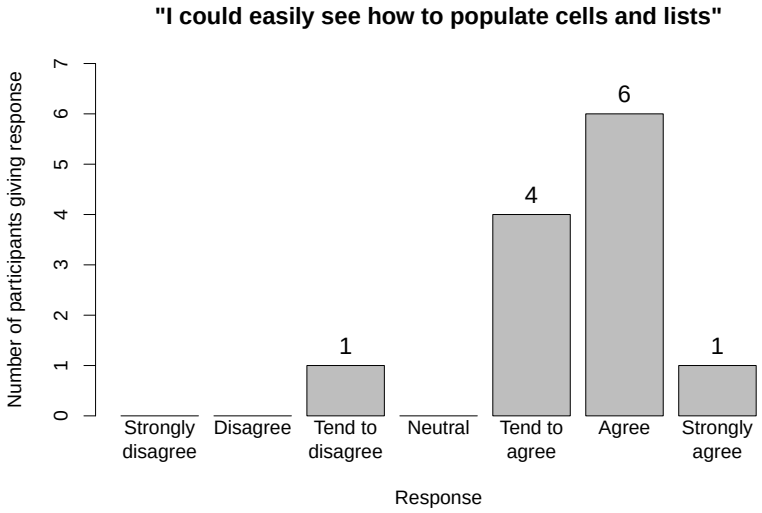


Figure 8.3: Responses to the question, "I could easily see how to create and populate cells and lists"

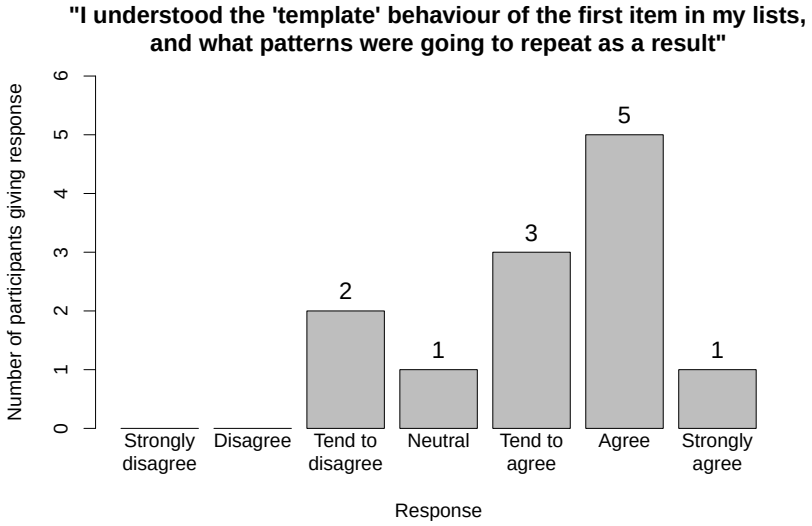


Figure 8.4: Responses to the question, "I understood the 'template' behaviour of the first item in my lists, and what patterns were going to repeat as a result"

and the answers imply that most users found the abstraction gradient of the lish fairly gentle.

There was much less consensus on the statement, "From the available building blocks of cells and lists, it was clear how to structure my model" (Figure 8.5). Five participants were either neutral or disagreed to some extent, while seven at least tended to agree. This question was about closeness of mapping: were participants easily able to translate the model they had in their head into its cells-and-lists representation? A couple of participants made comments later on in the interview which suggest that mapping between lists and tables is not necessarily trivial:

> "Your brain thinks it's a spreadsheet and wants it to behave as one, and when it doesn't it's slightly surprising." – *Participant W*

> "It's that concept of lists. I was still thinking very much in terms of rows and columns." – *Participant G*

From these comments and from the mixed answers to the question of Figure 8.5, it appears that closeness of mapping was a barrier, for some participants at least.

There was a much higher level of agreement with the statement, "When viewing and navigating the model, I could tell what the underlying list structure was" (Figure 8.6). This suggests that the lish scores well on role expressiveness, even though the typeset tabular representation does not, on the face of it, look very like a list of lists. The overlaid boxes that can be turned on in the editor, to make the nested structure more explicit, seemed to have helped the users here. The caveat mentioned above about the "compressed" empty template cells should be borne in mind in this context, however.

There was a strong divergence of responses to the statement, "When I wanted to insert the extra site, I could easily locate the cursor appropriately for the insert operation" (Figure 8.7). Five participants strongly agreed, one strongly disagreed, and the remainder were spread out in between. This may well have been a learning effect: once a user has grasped how insertion works at different levels they might find it very easy indeed, but not all participants had yet reached that point. This question explores another aspect of role expressiveness: does the interface provide the necessary beacon that says, "go here to expand the structure"? In this case the role being expressed is

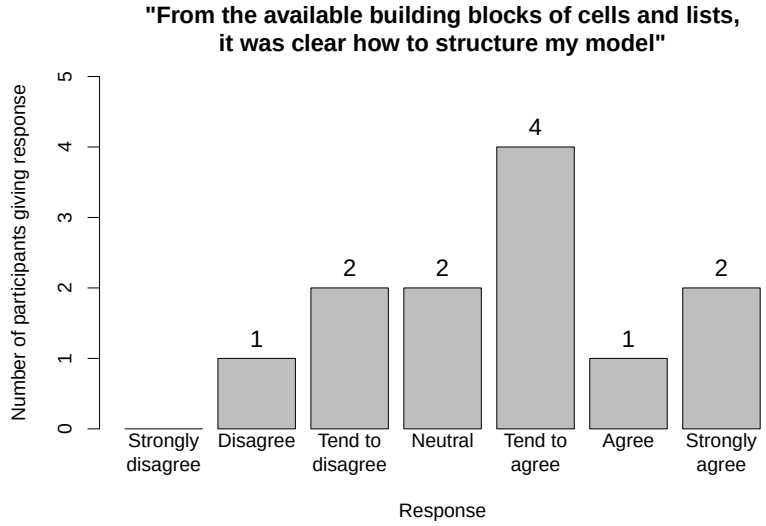


Figure 8.5: Responses to the question, "From the available building blocks of cells and lists, it was clear how to structure my model"

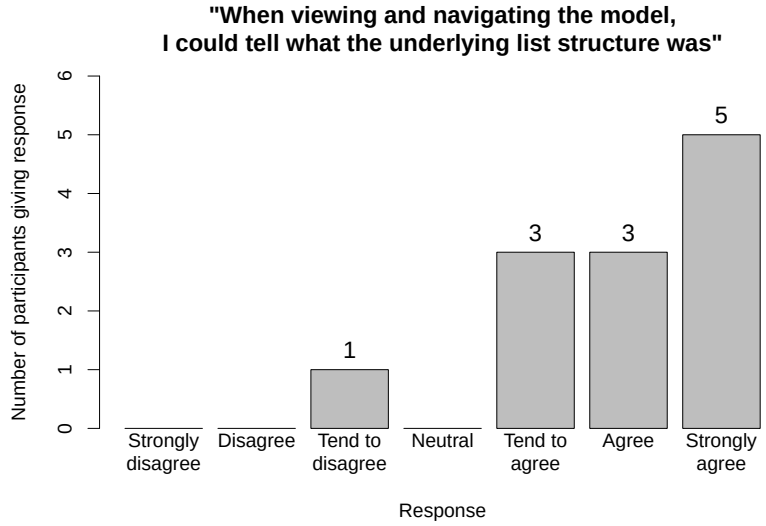


Figure 8.6: Responses to the question, "When viewing and navigating the model, I could tell what the underlying list structure was"

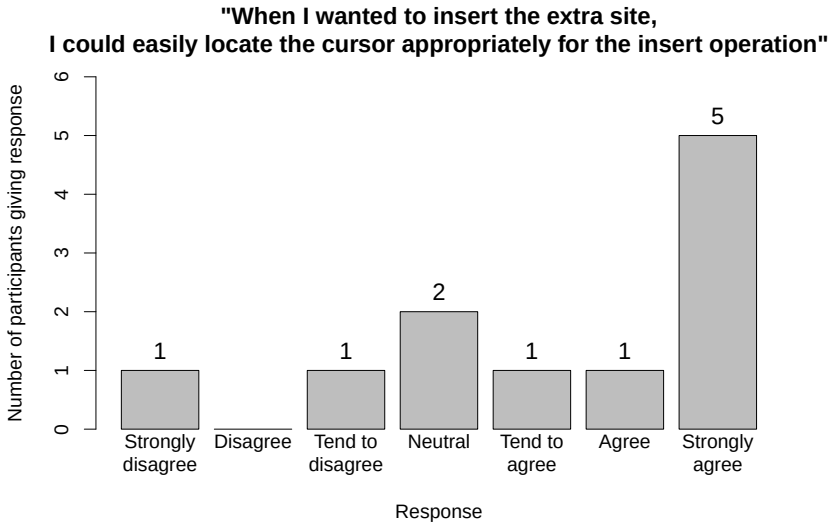


Figure 8.7: Responses to the question, "When I wanted to insert the extra site, I could easily locate the cursor appropriately for the insert operation" (Note only 11 responses, as one participant did not reach this part of the task)

that of an adjustment point, as opposed to the status of an object within an existing structure.

8.3.4 How easy was it to effect the calculations?

There was unanimous agreement with the statement, "I could easily locate where to put my formulae so that the correct *output* cells were selected" (Figure 8.8), but a more equivocal response to, "I could easily find the right location such that the correct *input* cells were selected" (Figure 8.9). These responses may have been coloured by wider difficulties in getting the structure correct so that a suitable input range even existed:

> "I think the structure of how you build it has a lot of impact on the inputs, so you have to kind of think in advance where you want to be going with it." – *Participant A*

The compressed empty template cells also appeared to be a culprit, as these were less conspicuous when the user was hunting around for a cell that would cause the desired input range to light up.

All participants agreed with the statement, "When performing 'vectorised' calculations, I understood the relationship between the formulae

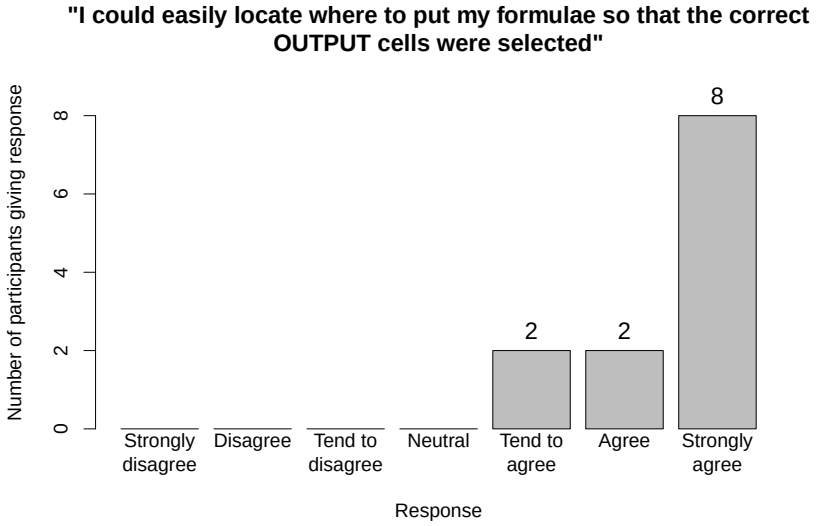


Figure 8.8: Responses to the question, "I could easily locate where to put my formulae so that the correct *output* cells were selected"

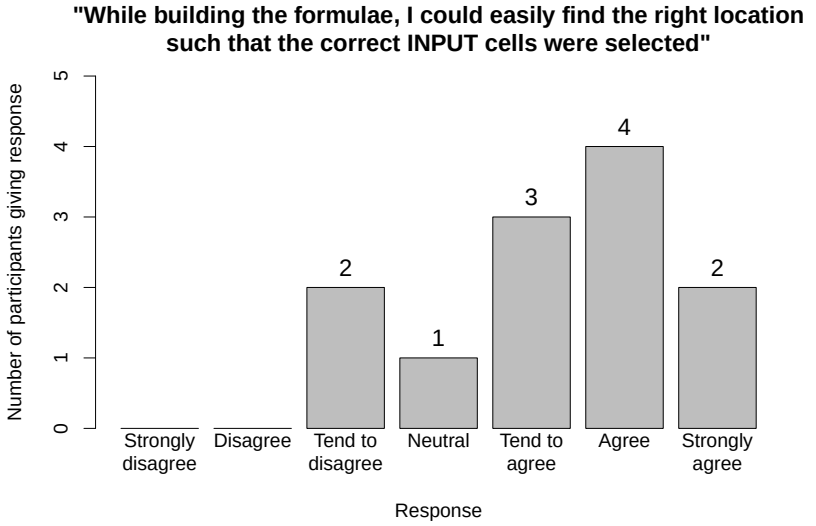


Figure 8.9: Responses to the question, "While building the formulae, I could easily find the right location such that the correct *input* cells were selected"

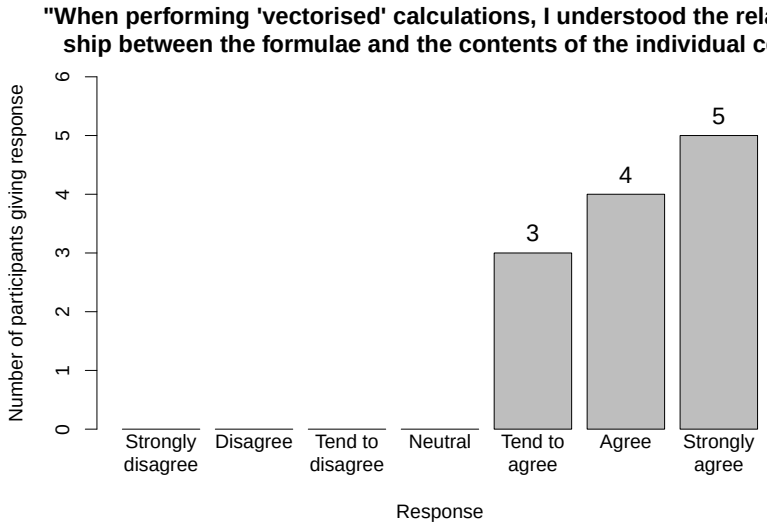


Figure 8.10: Responses to the question, "When performing 'vectorised' calculations, I understood the relationship between the formulae and the contents of the individual cells"

and the contents of the individual cells" (Figure 8.10). This is reassuring as one of principal aims of lish is to banish the formula replication of the spreadsheet. Most participants had at least a passing familiarity with the R language, so vectorised arithmetic was not an unfamiliar concept. The final formula in the task required the use of ranges of non-adjacent cells (for example, "all the managers", in Figure 8.2), but participants seemed to have no difficulty in conceptualising these as "vectors", even though the relevant cells were physically dispersed in the display.

The majority of participants agreed that "When I specified a 'sum' operation, the machine interpreted it the way I expected from the shape of the layout" (Figure 8.11). The lack of full agreement was likely to have been related to the instances of circular references observed above, as in these instances the behaviour of the 'sum' formula would have appeared counter-intuitive.

8.3.5 Relating the lish to business as usual

The questions up to this point had all been about the small example model that featured in the task. But how do we know that this model is relevant to the actual work of analysts in the wild? Even if the lish can improve

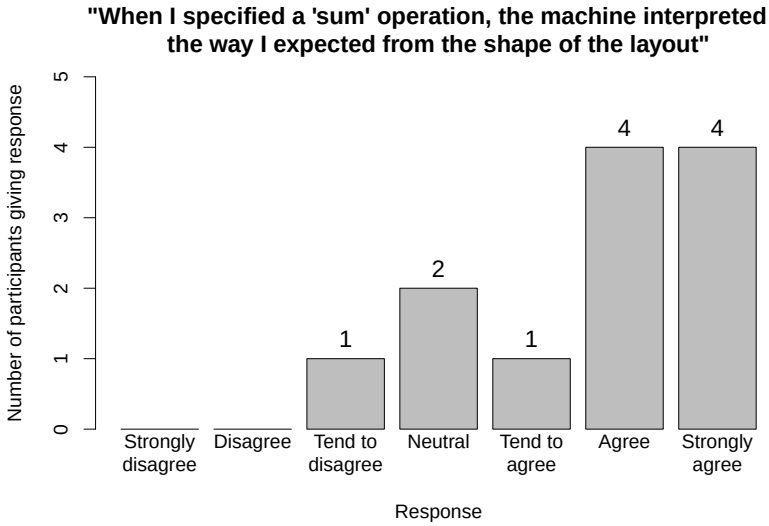


Figure 8.11: Responses to the question, "When I specified a 'sum' operation, the machine interpreted it the way I expected from the shape of the layout"

certain aspects of a certain analytical workflow, are they the right aspects of the right workflow?

The final block of questions sought to complete this missing link by asking participants to relate what they had seen to their own business as usual (BAU). In the first of these questions, the participant was asked about the prevalence of lish-like structures in their own work. This question sought to assess whether the structure itself, irrespective of the facilities provided for processing and interacting with it, bore a close resemblance (closeness of mapping, again) to the structures encountered in real-world models.

This was an "open" rather than multiple choice question. All participants responded that lish-like structures were at least fairly prevalent, and in some cases almost universal, in their work (they were not asked for specific examples due to confidentiality considerations). Five mentioned unprompted at this question, or elsewhere, that they thought the lish would make quality assurance (QA) easier. There was one contrary view: that the person doing the QA might not understand how the lish worked, so it could actually make QA harder!

The next question asked about trade-offs between the lish and other tools such as spreadsheets that the participant might use in their BAU. The question was framed in this way so as to encourage participants to be open

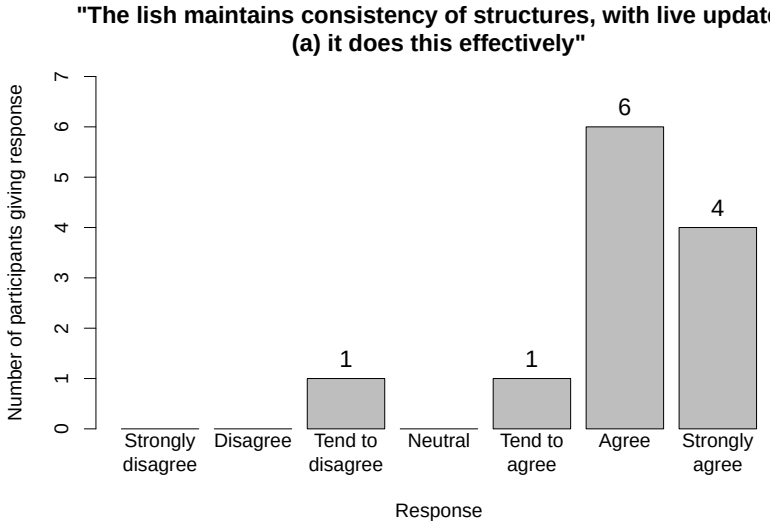


Figure 8.12: Responses to the question, "The lish maintains consistency of structures, with live updates: (a) it does this effectively"

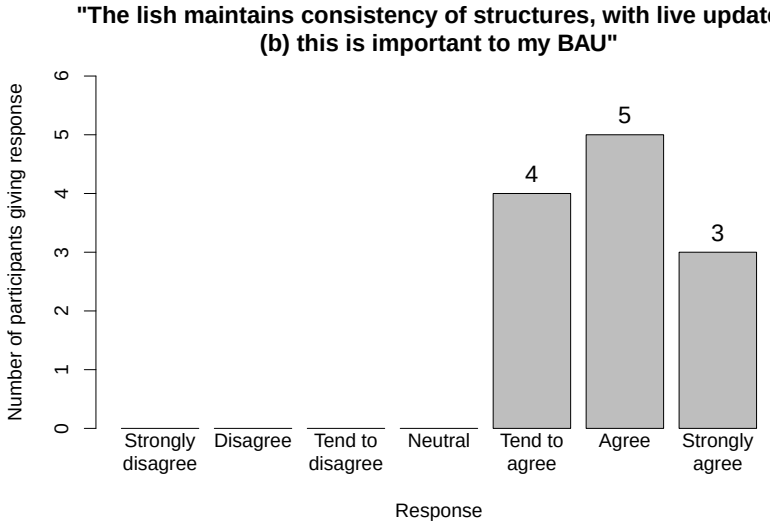


Figure 8.13: Responses to the question, "The lish maintains consistency of structures, with live updates: (b) this is important to my BAU"

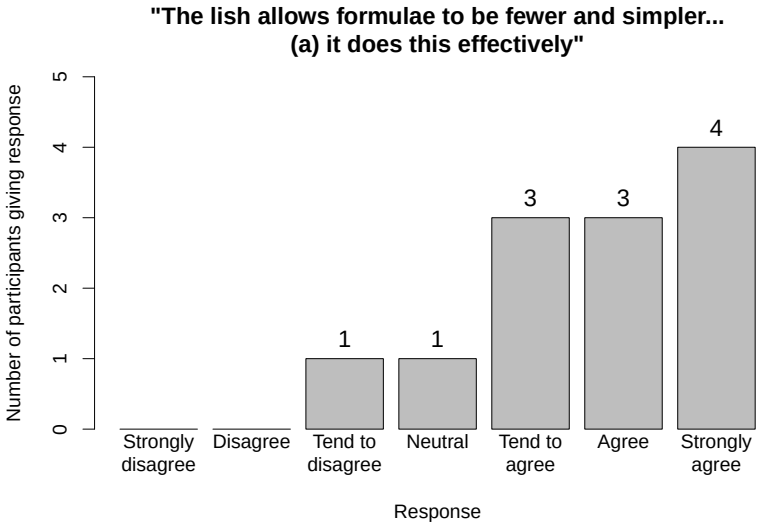


Figure 8.14: Responses to the question, "The lish allows formulae to be fewer and simpler (compared either to a spreadsheet, or to expressions you might have to compose in a script): (a) it does this effectively"

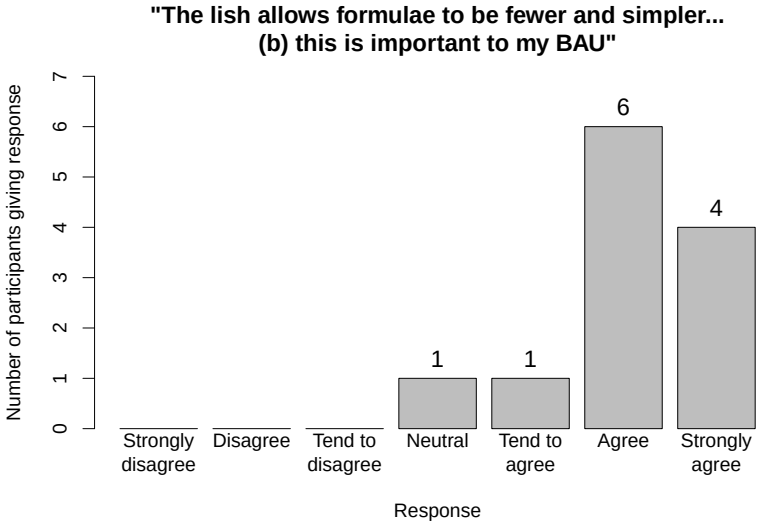


Figure 8.15: Responses to the question, "The lish allows formulae to be fewer and simpler (compared either to a spreadsheet, or to expressions you might have to compose in a script): (b) this is important to my BAU"

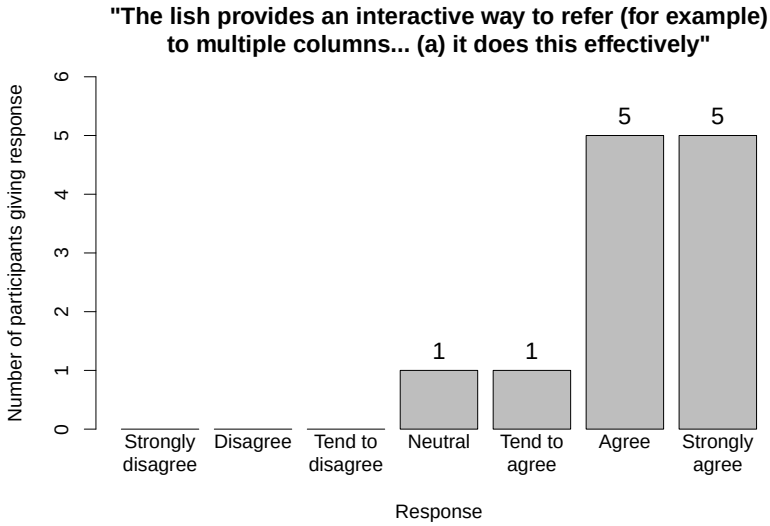


Figure 8.16: Responses to the question, "The lish provides an interactive way to refer (for example) to multiple columns, from separate but related tables: (a) it does this effectively"

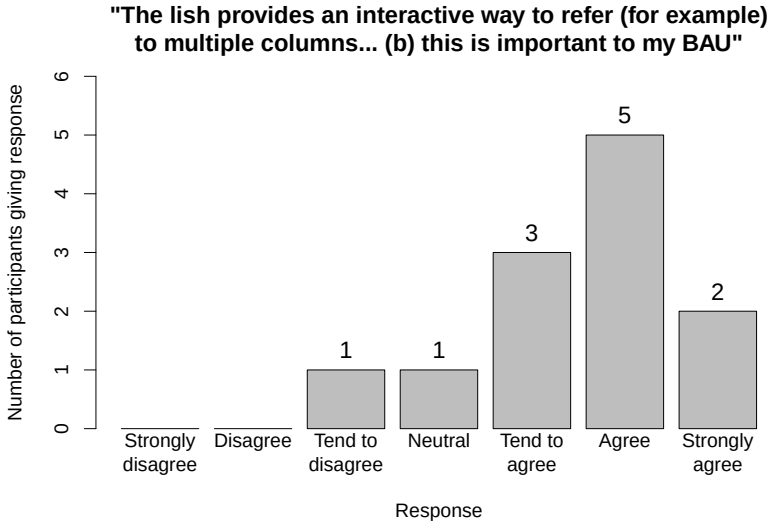


Figure 8.17: Responses to the question, "The lish provides an interactive way to refer (for example) to multiple columns, from separate but related tables: (b) this is important to my BAU"

if they perceived any of the apparent advantages of the lish as coming at the expense of weaknesses that would undermine its use. There were some discretionary interviewer prompts on the potential trade-offs between upfront effort in building a model versus later payback in extending or maintaining it; and between keeping the building blocks simple versus having to assemble higher structures from scratch.

Due to time constraints, these additional prompts were not used with all participants. All participants who did discuss these points perceived there to be significant upfront effort to form the structure, but considered this a worthwhile trade-off (at least for larger models) for the advantages of having that structure once in place. Two mentioned that they would use it for larger models but not for quick ad hoc work. Some comments give a hint as to the root cause of the need for effort upfront:

> "The main limitation is flexibility if you need to change things last minute, but worth it for the analysis it can drive." – *Participant V*

> "I perhaps normally tend to just throw the data in and then fit a structure around it, rather than the other way round." – *Participant A*

This need to plan in advance can be seen as a manifestation of *premature commitment* – the user cannot simply "throw the data in" and create the structure afterwards. It is a shortcoming that is driven in turn by *viscosity*, which arises from the difficulty of refactoring a lish structure if the user's initial attempt was not quite correct. This rather discourages the user from a trial-and-error approach. Another participant considered this to be a positive benefit, though, as it forces the user to be methodical:

> "It front loads the thinking and structuring a little bit which is a good thing." – *Participant C*

All participants who expressed a preference were for constructing their models from the simple building blocks provided, as opposed to from more specialised higher level objects. The better understanding of one's own model that can arise from having built it from scratch, and perhaps the extra flexibility also, seemed to win the day:

> "The trade-off again is almost a didactic one... if you give them a table button then they will approach it as a table as opposed

> to a list of lists so it's a kind of longer term pay-off to build from basic building blocks... [otherwise] people would not understand the potential power of using it like that." – *Participant C*

The last three questions looked at three of the main ways in which the lish tries to support the user, namely:

1. Providing live updates to structure as well as to calculated values, thereby maintaining consistency in the model.

2. Reducing the number of formulae, and simplifying their form.

3. Selecting cell ranges interactively, including non-adjacent ones.

For each of these three aspects, the participant was asked to give separate scores on two dimensions: whether the lish supported this aspect effectively, and whether the aspect was important to their BAU.

On providing live updates, the vast majority (11 out of 12) agreed with the statement that the lish does this effectively (Figure 8.12). The one dissenter pointed to the problem that might occur if a data supplier changed their format. One other person mentioned this as a caveat but still agreed that the lish was effective in this regard. All agreed it was important to their BAU (Figure 8.13).

On simplifying formulae, 10 out of 12 agreed that the lish does this effectively (Figure 8.14), and 11 out of 12 that it was important to their BAU (Figure 8.15). Three participants mentioned that the benefit might be more questionable if comparing with a coding language rather than a spreadsheet.

On interactive selection, 11 out of 12 agreed that the lish does this effectively (Figure 8.16), and 10 out of 12 that it was important to their BAU (Figure 8.17). Participant Q commented that this was "one of its strongest points"; Participant C described this aspect as "definitely very strong. Would be a way to save time, and if you had a large model could be very powerful".

8.4 Limitations and future work

In this section, I reflect on the limitations of the current study and consider some options for improvement in future iterations.

As was noted at the beginning of the chapter, participants were drawn from a limited population (GORS analysts). It would clearly be desirable to test the lish with a wider range of participants, and with a larger sample size. One feature of the study sample that is a particular limitation is that most participants had had some exposure to the R programming language, and hence to the notion of vectorisation in an analytical environment. Since one of the features of the lish being tested by the study was its vectorised arithmetic, these participants had a head start compared to the population of spreadsheet users at large.

What are the implications of the head start? We might consider the assessment of vectorisation in two stages. First, do analytical users who have not met vectorisation before comprehend it as easily as they would iteration over scalars? And second, once they have the advantage of such exposure, do they readily grasp the way it is extended by the lish? In particular, this includes the way in which certain cells, although visually dispersed in the display, can nevertheless behave as a single "vector".

The study as it stands addresses the second question, but not the first. Furthermore, the group targeted by the sample (consisting of expert analysts) might be expected to contain proportionally more users than the wider spreadsheet-using population for whom the answer to the first question is "yes". The scope of inference would be greatly enhanced, therefore, if a follow-up study could expressly test the lish with users not having the advantage of prior exposure to vectorisation.

One hazard of any study where the participants know the researcher is that it is difficult to test questions along the lines of "How good is my novel system at…?" without fear of bias[1]. I sought to mitigate this problem with balanced wording that actively encouraged participants to comment both on strengths and on weaknesses; many did so. Nevertheless, there is a risk that overall, the results present the lish in an over-optimistic light. The reliability is rather better in eliciting on a *relative* scale which features of the lish participants found easier to understand than others. For example, we saw that not all users found it clear how to structure their model (Figure 8.5), whereas they generally did find it clear from the display what the underlying list structure of an existing model was (Figure 8.6).

There is another more subtle form of bias that might arise out of the choice of questions asked. As noted in section 8.2.3, the questionnaire tar-

[1] Presumably in a positive direction, assuming the participants also *like* the researcher.

geted those cognitive dimensions that seemed (to the researcher) most relevant to the lish, especially where a desk-based exercise lacking user involvement was inadequate. But as Blackwell and Green [2000] point out, there is a distinct risk that this approach will screen prematurely which cognitive dimensions the users get asked about. It might turn out that the interview designer has overlooked aspects that are actually very important to users. A future study might address this problem by using the generic cognitive dimensions questionnaire that Blackwell & Green [2000] have designed. This questionnaire uses wording that does not suppose any prior knowledge of the dimensions, and allows the participants themselves to identify which of the dimensions are relevant and important to them.

A further measure that could be taken to improve future studies would be to apply the *cognitive walkthrough* technique to the design of the study. This technique (as applied to user interfaces) originated with the work of Polson et al. [1992]. It is based on a hand simulation of the cognitive activities of the user as they perform the sequence of actions necessary to communicate with the interface, from the point of view of a new user who is exploring the system. It may be carried out at the design stage before any implementation has been produced, and hence identify problems upstream of committing development effort.

The cognitive walkthrough technique was further developed by Green et al. [2000], who applied it to the design of user studies. Their insight was that it can easily happen (especially with prototype systems) that difficulties experienced by study participants occur accidentally, due to aspects of the system quite separate from those intended to be the subject of the evaluation. They therefore extended the walkthrough to pre-check all the stages of the activity that the participant is going to be asked to perform. This includes interacting with the user interface, as before, but also the broader reasoning and knowledge necessary to complete the task. Hence potential difficulties with the task that are extraneous to the aspects under evaluation can be identified in advance and remedied. Unnecessary noise in the results is thereby avoided.

Both flavours of the approach would be applicable in evaluating the lish. The reach of the desk-based critique could be extended following Polson et al. [1992], and future user studies could be made more robust following Green et al. [2000]. Indeed some elements of the latter approach were applied informally to the present study: I conducted a "test drive" at the

planning stage which allowed me to identify and amend some features that felt awkward to use. But the more exhaustive enumeration of the steps involved in a full walkthrough stands a better chance of noticing difficulties that the lighter touch treatment might miss. In future studies, therefore, it would be beneficial to apply this as a screening technique before committing to live users.

8.5 Conclusion

The study gave empirical support to the expected outcome that "wide" data are more human-readable than "long", at least among the limited population that was sampled.

Most participants were able to complete a small task using the lish, but all required at least some hints. So how to use the lish is not obvious "out of the box". The comments by participants suggested that lack of experience, as opposed to conceptual misunderstanding, was the biggest barrier.

Both the observations of the task and the questionnaire responses afterwards suggested that the lish has a low abstraction gradient, but sometimes falls short on closeness of mapping between the list-of-lists structure and the user's mental model. Role expressiveness was better than expected from the not very list-like appearance of the lish as typeset in the editor, but the high viscosity of the lish was problematic, and led to participants feeling they needed to plan in advance rather more than with a spreadsheet. The vectorised formula model was generally well understood (at least in this population, who had typically had some exposure to vectorisation before), but creating the supporting structures required to use it effectively was sometimes more difficult.

The caveats of the previous section regarding potential bias should be borne in mind with the more subjective questions. If we take the responses at face value, however, all participants were of the opinion that the structure of the lish matched the patterns they observed in the models they dealt with in their business as usual, at least a high proportion of the time. There was a decided preference (in this user population) for the lish approach of providing simple building blocks that could be composed into a wide variety of more complex structures, as opposed to providing numerous higher level abstractions to model those structures directly. The lish sets out to make specific contributions to the state of the art in certain areas: live structural updates, simplification of formulae, and the novel range selection method.

Strong majorities of the participants agreed both that the lish approaches to these areas were effective, and that these areas were important to them within their own work.

Chapter 9

A review of the lish

In this chapter I begin by returning once again to the three original research questions, and offering some answers. I then reflect more broadly on the suitability of the lish as a model for spreadsheet-like tabular data, including a frank assessment of where it falls short. I finish by setting out a plan for how these shortcomings might be addressed.

9.1 The research questions in review

9.1.1 RQ1: How can we model spreadsheet-like data using a representation based on lists of cells?

Taken alone, this question almost begs the rejoinder, "but why would we want to?" The full implications of adopting lists of cells, after all, were scarcely foreseeable in advance. But some advantages were clear early on: the ability to capture hierarchy, and to provide an explicit boundary around groups of cells that are to be treated as a single object.

Having decided to entertain lists of cells as a possibility, one immediately runs into the problem that the list on its own, as a one-dimensional object, cannot express tabular constraints. The nature of the problem and the solution I adopted were summed up in Figure 3.4 and developed in detail in Chapter 4. By conferring a "template" behaviour on the first element of each list, I created a model that could capture repeating cell patterns. This not only recovered the tabular behaviour, but enabled a much richer family of repetitive structures to be expressed.

A second problem, though a less fundamental one, was how to handle the geometry when progressing from lists in square brackets on paper to a two-dimensional display resembling a spreadsheet. This was the subject

of Chapter 5, where the notion of *compound indices* was a key device in identifying elements to be aligned.

It would have been feasible for the lish to retain spreadsheet-like cell references in formulae, perhaps using some kind of nested indices instead of A1-style referencing. But the highly structured data model of the lish was a more natural fit for a vectorised style of formula expression, which I developed in Chapter 6.

The lish, then, provides an answer – one answer – to the "how?" The other research questions concern the "why?"

9.1.2 RQ2: What would be the consequences for analytical workflow of using a lish, instead of a grid, as the basis of a spreadsheet-like environment?

In an ordinary spreadsheet, application structure is implicit in the spatial arrangement of cells on the grid, and formulae are defined over user-specified ranges corresponding to objects of interest. In a lish, the structure is explicitly expressed by the sublists, and formulae are defined over the inheritors of template cells.

As was observed in the literature I reviewed in section 2.3.3, there are several advantages to being able to decompose a spreadsheet application into cognitive units [Hodnigg and Pinzger, 2015]. One essential difference between the grid and the lish is that in the grid, this decomposition must be inferred (with the potential for ambiguity), whereas in the lish it is already present in the structure. An immediate consequence is that the lish is able to provide its interactive range selection mechanism, which was popular with users in the study. This offers not only a convenience but a reduction in the risk of out-by-one errors, because cell selections correspond to meaningful objects (such as a column group) rather than to arbitrary ranges of cells.

The other major difference from the grid is in how formulae are handled. Lish calculus begins from the notion of vectorised expressions, but makes them more general than regular vector and matrix arithmetic. For example, in the materials lab example, the sample mass was treated as a single "vector" even though its cells were dispersed throughout the lish. The vectorised notation avoids the need for busy expressions along the lines of "find the sum of all the cells that are the third element, of each list that is a sublist of such and such". In the quantitative taxonomy of Panko and Aurigemma [2010] this helps to address spreadsheet planning errors; and in particular,

ranges that cut across dimensions address the problem of non-2D logic.

Vectorised formulae typically act on whole objects, as in `sum($sales)`. This enables formulae to be very DRY[1]. So in the qualitative taxonomy of Leon et al. [2015] they address formula integrity (because potentially inconsistent multiple formulae are eliminated) and extendibility (because formulae are robust to adding extra rows, or even extra dimensions). Several participants in the user study picked up on the quality assurance benefits of this approach.

Furthermore, the notion of archetypes which can be matched between formula arguments reduces the amount of information the user has to give, because the machine can usually deduce without being told at what depth recycling or aggregation is intended to occur. The presence of hierarchy in the model allows us to dispense with the traditional spreadsheet's distinction between relative and absolute references and instead use the "most locally appropriate" value (subsection 7.4.3).

Workflow is also influenced by the cloning mechanism, which promotes a style of working in which the user can build from the particular to the general. Unlike the copy-paste equivalent in the spreadsheet, clones in the lish refrain from duplicating their associated formulae, so DRYness is not undermined.

Participants in the user study recognised these contributions to building analytical applications as both effective and relevant to their business as usual. But do these benefits come at a cost of raising the cognitive barrier to using the lish? In the third research question, I evaluate this aspect.

9.1.3 RQ3: Where would this alternative representation be located in the space of the cognitive dimensions?

I will follow the categorisation of the cognitive dimensions that I gave in section 2.4.

The dimensions of structure

(Abstraction gradient, Closeness of mapping, Premature commitment and *Viscosity)*

The abstraction gradient of the lish, as represented in the editor, may reasonably be said to be low because it is built from so few parts. Once the user understands cells, lists and templates, they have all the basic building

[1] "Don't Repeat Yourself" – see subsection 2.3.4.

blocks. In the user study, users readily grasped these. There may be a price to pay for this simplicity in terms of closeness of mapping. Although the applications I have considered could all be represented in lish form, there was sometimes a need to play "programming games" – for example, most users in the study had difficulty with the need to form a column group before the sum they needed could be calculated. I shall return to closeness of mapping in more depth in subsection 9.2.1.

Premature commitment is low in the sense that a user may create a rough prototype of an object in advance of a full design, and then later decide to clone it. Viscosity in this scenario is also low: the prototype may be changed even *after* it has been cloned, and all the clones will automatically update as well. Similarly the lish makes it straightforward to add extra levels to a dimension, and extra dimensions to an application.

On the other hand, several users commented on a need to plan in advance more than was the case for the spreadsheet, which suggests they perceived premature commitment to be higher. The programming games seemed to be one cause of this; high viscosity to certain types of refactoring (as in the example of spanning columns, section 7.3) might be another.

The dimensions of visualisation

(Diffuseness, Hidden dependencies, Role expressiveness, Secondary notation and *Visibility)*

The current visualisation of the lish tends to be rather diffuse for structures with more than two dimensions, due to the proliferation of template cells. A future version of the editor needs to provide a view in which these are hidden altogether. The lish does rather better on diffuseness when it comes to formulae though: there tend to be far fewer of them than in an equivalent spreadsheet, and the vectorised notation makes them concise.

The lish shares the same problem as the spreadsheet with hidden dependencies: the user cannot tell by inspecting a cell which other cells depend on it. Just as in the spreadsheet, this could be mitigated by an audit tool. The lish has the further problem that the value in a cell may be the result of a formula that is *not* in that cell: the price of DRYness. Once again, however, a suitable audit tool could readily disclose this dependency. The lish departs somewhat gracefully from the behaviour a spreadsheet user might expect here, in that there is a clear hierarchy of ownership of cells. A cell in a lish cannot acquire a value as an arbitrary side effect of a calculation

somewhere else, but only when an owner in a well-defined chain of template cells has imposed it.

The user study found (subsection 8.3.3) that role expressiveness in the lish was effective in terms of users being able to tell from the display what the associated list structure was. In terms of locating the correct adjustment point at which to execute an operation it was weaker, however.

The current editor has limited support for secondary notation, but supplies by default visual cues such as gridlines and shading that might be secondary notation for a typical spreadsheet user. An interesting future direction would be to try to align the creation of structure more closely with the actions of adding these visual characteristics, to make the process of building a lish feel more analogous to formatting tables in a normal spreadsheet.

As for visibility: in common once again with the spreadsheet, the current editor prefers visibility of data over visibility of formulae. Since the lish has far fewer formulae than the spreadsheet, it would be comparatively straightforward to create a lish view in which both data and formulae were visible.

In this project I have only considered small applications that fit largely within a single window. A future improvement that would help the lish to scale much better to larger datasets would be to limit the display rectangle of each sublist and allow scrolling *within* this rectangle, as well as for the whole window. This would let the user see more of the structure at once. The idea is similar to "freeze panes" in a spreadsheet, but at a local, sublist level. Since the lish already "knows" which cells are titles and which form a table body, it would be a natural fit for this type of visualisation.

The dimensions of calculation

(Consistency, Error proneness, Hard mental operations and *Progressive evaluation)*

Consistency in the lish is helped by the small number of concepts and rules (see abstraction gradient, above). There is only a little for the user to learn, so only a little in terms of scope for unexpected or exceptional behaviour. In the formula language, the uniform treatment of different operators (section 6.5) helps with consistency.

The lish uses vectorisation instead of explicit iteration, and range selection is by means of template cells. Although it would require a much larger

user study than the one in this project to test empirically, it is reasonable to suppose that both these features offer some protection against careless errors, or slips – in particular, with regard to "out by one" errors. This cannot, of course, address logic errors!

With regard to hard mental operations, simple table structures are easy for a user to learn, but they do not have to go very far before the programming games become a barrier – as witnessed by the mixed responses from users to the question about the difficulty of structuring the model (subsection 8.3.3). Other aspects seem to alleviate hard mental operations, though. The formulae used in the study were rather simple so did not exercise this dimension fully. But I note here that in more complex applications, lish formulae remove the need in most cases for the user to specify explicit information about what level they want to refer to within the structure. This may reasonably be expected to make the associated operations easier.

Progressive evaluation is straightforward: like the spreadsheet, the lish has it.

9.2 Discussion

In this section I return in a little more depth to some aspects of the lish that can be identified as root causes of those areas where it works well, and those where it does not.

9.2.1 Mathematical versus cognitive mapping

The original rationale for replacing the spreadsheet grid with the lish was that the lish might better capture the structure present in the user's application; indeed, this was the prerequisite for any other benefits. Out of all the cognitive dimensions, the one directly targeted by this approach is closeness of mapping. I now therefore revisit this dimension and ask, just how close is it?

In subsection 9.2.2 I shall pose the related question of *expressiveness* – can the lish represent all the structures that might be of interest? In this section, I shall stick to the cognitive side – is it easy for the user to see how to map a given structure from their application into a lish?

Consider the following list of characteristics known to occur in spreadsheet data. In this dissertation, we have seen examples of all of them. (Several of the characteristics are similar or overlap in some way, though

none are entirely equivalent.)

hierarchy – where an object contains high and low level, or external and internal, structure.

repetition – where the same pattern of cells is to appear in more than one location.

templates – a form of repetition where some properties of an object are fixed by a template, but others may vary per instance.

partition – where data are to be split into non-overlapping portions.

grouping – similar to partitioning, but with an implied subordination of grouped objects to ungrouped.

coordination – where the Cartesian product of two or more sets of objects generates a third set.

ownership – a form of $1 : n$ relationship, between 1 owner and n owned objects.

inheritance – where the properties of an object are defined by another object with which it has an "ancestor" relationship.

cross-cutting – where an object is composed from equivalent parts taken from different branches of a hierarchy.

The lish is able to model all of these characteristics, so in a *mathematical* sense it has excellent closeness of mapping. But in a *cognitive* sense, that is not necessarily so. Lists and 2D tables are simple enough; cloned objects, while conceptually following a rather obviously list-like structure, seemed less simple to the users in practice (Figure 8.7). Programming games, such as the column group "trick" in the user study and the construction of spanning headings, are clear examples where the cognitive mapping is not as close as we would like, even though the mathematical mapping cannot be faulted.

The problem the lish has in these situations is the very limited vocabulary at its disposal. Everything is either a cell or a list, and first elements always repeat. This is excellent for reducing the abstraction gradient, but may be a form of *primitive obsession* [Fowler, 1999]. Whatever structural feature of the data we want to model, the lish's answer is to add yet another level of nesting! This is perfect when the feature in question is hierarchy,

but falters when it is partitioning, or adding a dimension, or ownership. Although nesting can faithfully represent these features also, they require the user to join a few more mental dots. In the example of Figure 7.4, the lish was nested nine levels deep, so keeping track of which level performed which role would likely be challenging for a user, as well.

But before we rule out the lish in favour of a more highly featured model, we should beware of merely shifting the cognitive load elsewhere – in the form of an increased abstraction gradient – rather than reducing it. The users in my study expressed a preference for simple building blocks over an abundance of more specialised high level data types. How, then, can we keep the model "as simple as possible, but no simpler"? The answer, I propose, is to make *some* of the structural characteristics on my list above first class properties of a lish-like model – but certainly not all of them! The new research question becomes, what is a sufficient minimal subset of these characteristics for a model that is as expressive as the lish while simplifying (or ideally eliminating) the programming games?

9.2.2 Is the lish expressive enough?

We saw in the literature of subsection 2.3.7 and in the sample density example of section 7.4 that hierarchy is an important concept not captured by the spreadsheet grid. So the ability of the lish to express hierarchical structures as well as flat tables is, even by itself, an advance. And we have seen that the lish can also express a rich assortment of structures: tables, arrays (of arbitrary dimensions), column groups, spanning columns, sequences of similar tables, and other predictable repeating patterns like the sample density.

An important enabler for this flexibility was sacrificing the number two as a privileged number of dimensions. Discarding the 2D grid was a liberation rather than a loss, because once the template rule had recovered the ability to express 2D behaviour came the realisation that the same rule could express all the other structures as well. Formulae could then be not only per column, but could populate a whole table or any of the patterns of repeating cells, e.g. "all the sample masses". So scaling an application to add an extra dimension became straightforward.

The lish is also flexible enough that we don't need a separate abstraction for a "canvas" or similar in which lishes are to be embedded for the calculations. The lish supplies the canvas as well, so there's not a parallel

set of operations for the user to learn when it comes to inserting, deleting and editing items.

The study participants were of the opinion that the lish was a suitable structure for the applications they encountered in their business as usual, but a limitation is that I have not carried out a comprehensive survey of spreadsheets to look for patterns that the lish *cannot* express. It would therefore be presumptuous to suppose that it can model all the forms of data that might appear in existing spreadsheets, and indeed I have already observed a couple of counterexamples.

First, there is a limitation on the repeating patterns: it is always the first element of each list that repeats. Sometimes the user might want a repeating cell pattern that is not based on all subsequent elements of a lish conforming to the first – for example, a class of object that is reused, but not always within the same list. This was a conscious trade-off, influenced by relational database design where one would usually (but not inevitably) store entities of the same type in the same table. While less flexible than arbitrary repetition, it avoids the user having to make explicit links between the template and its instances, and hence introducing a further hidden dependency. The alternative design decision would be interesting to explore, however.

Second, there is a limitation that I call the "meta meta data" problem. Suppose we have some column headings that are month names, and these reside in a lish whose first cell is labelled "month". We might wish to set all the column headings in bold font. If we set this property on the month cell it will be inherited by the individual months as intended, but also unfortunately by every value in the columns they own. There is no way to stop inheritance propagating without overriding it in multiple places. To address this problem, the lish needs to treat metadata more explicitly as a separate dimension. For the example just given, the current editor does provide distinct properties called `font_weight` and `label_font_weight`, but this workaround feels a little clumsy.

9.2.3 On tables as lists

The original motivation for the lish's template behaviour (Figure 3.4) was to enable nested lists to represent "proper" tables, where each row is constrained to contain the same number of cells, and column insertions respect the integrity of the table. We saw later that the lish is (ostensibly) agnos-

tic to whether the table is arranged as lists of rows containing one cell per column, or vice versa.

The inheritance mechanism indeed confers some symmetry, in that whichever arrangement is chosen, row-wise and column-wise selections are both possible. This is as it should be, because when the user has decided that the object they want is a 2D table, its underlying list representation should intrude as little as possible.

Some departures from the ideal begin to appear in the way the user has to build and navigate the structure. The basic operations on the lish are defined in terms of adding and enclosing cells in lists, and the cell cursor may select a cell, a list at any level, but not (say) a column, if the table is a list of rows. So even when working with tables, the user must still at some level keep the list representation in mind. There are other oddities; consider the example below where some rows have been grouped:

$$\left[\begin{array}{llll} [& \text{Id code}, & \text{V1}, & \text{V2} \quad], \\ [& \text{id1}, & 1, & 2 \quad], \\ \left[\begin{array}{l}\text{Group A},\\ \text{idA1},\\ \text{idA2}\end{array}\right], & \left[\begin{array}{l}\bullet,\\ 3,\\ 5\end{array}\right], & \left[\begin{array}{l}\bullet,\\ 4,\\ 6\end{array}\right] &], \\ [& \text{id2}, & 7, & 8 \quad] \end{array}\right]$$

The top level table is arranged as lists of rows, but inside the group we must have a list of columns[2]. Again, cognitive closeness of mapping may suffer.

Some more serious cracks start to appear once we start composing templates, as in Figure 3.5. In the absence of composed templates, given any lish within the structure, one can always point to some other lish and say "there is its template". But with composition, the template has become a rather ethereal object, no longer visible to the user. Worse still, it cannot even be guaranteed to be a lish, so in subsection 4.3.1 I had to introduce a new abstraction, the trace, to represent it. A lish where templates are composed tends to be more brittle to transformation, increasing viscosity to the user. For example, the user might want to *transpose* the lish of Figure 3.5, so that rows become columns and columns become rows. But the result would actually no longer be a lish! The potential for such a simple transformation to break the model looks rather a serious shortcoming.

[2]There is in fact an alternative lish representation, using what I call an *auxiliary margin*, which retains row sublists throughout. But that requires another "programming game".

How can we make the data model more symmetrical within a 2D structure without conferring special privilege on the 2D grid, as the spreadsheet has done? In my list of structural characteristics on page 171, *coordination* is the one that allows any number of dimensions while treating them symmetrically. Of those characteristics listed that might become first class properties of a successor data model, this one therefore appears a strong candidate.

9.3 The future of the lish

In this section I summarise briefly the future research directions for the lish, or its successor data model.

At a simple implementation level, there are several improvements waiting to be made to the editor: the provision of an alternative view to hide template cells, the handling of labels when they appear in the results of formulae, and within-list scrolling, to name the major ones. And although I have made a pragmatic first attempt at visualising the lish, this would benefit from a more systematic approach to ensure that the underlying structure is conveyed to the user as clearly as possible.

A more substantial area of research lies in extending lish calculus. In my exploration in Chapter 6, I have so far covered only some simple functional forms: univariate functions, binary operators and aggregation functions. In section 6.9 I outlined a large area of future work to generalise further functional forms for the lish. An attractive early goal within this area would be the lish analogue of the relational table join. This would be an important further step down the road of allowing the user to visualise data in *untidy* form (in the sense of Wickham [2014]) while automatically recovering the tidy version for use by the machine.

As I have discussed in this chapter, however, the lish itself is not an entirely adequate data model. The immediate future work will therefore involve some fundamental rethinking to address the issues raised in section 9.2. What, then, will the successor to the lish look like? The short answer is, rather similar to the lish. As we have seen, the lish has many advantageous properties. The new model will retain a hierarchical structure, and hence at least a close analogue of the current lish calculus that enables formulae to be defined over it. It will retain a template and inheritance mechanism, since that was instrumental in providing the user with interactive range selections. It will differ however in that some of the structural characteristics

listed on page 171, to include *coordination* at least and possibly some others, will become first class properties. This is likely to require a modification to the existing template rule, and perhaps even a departure from a strictly list-based representation.

9.4 Overall conclusion

The majority of prior research on reducing spreadsheet errors has retained the two dimensional grid as the underlying data model, and built additional apparatus on top of or alongside this grid. I have argued for an alternative (or sometimes, complementary) approach, in which the grid itself is replaced with a data model that better reflects the structure of the user's data.

My grid replacement, the lish, is a minimalist model: in essence, nested lists of cells with repeating first elements. How far could we have expected to get with so few components? The answer, it turned out, was surprisingly far. The lish forms a natural basis for a flexible, template-based workflow, that ensures structural consistency. It incorporates a DRY, vectorised formula model, that can partially deduce the semantics of a calculation from the underlying structure. It supports a novel interactive method of selecting cell ranges. And the lish defines a flexible coupling between the data structure and the geometry as displayed, to combine the freedom of layout associated with a spreadsheet with some of the discipline associated with a database.

I evaluated the lish in the cognitive dimensions framework, with the help of some worked examples and a user study. The structural attributes scored well on abstraction gradient, consistency and role expressiveness, and the formula model contributed to reducing error proneness and alleviating hard mental operations. The very simplicity of the lish led, however, to some representations feeling awkward or contrived. The reliance on nesting of lists to represent multiple concepts undermined closeness of mapping; and the forced asymmetry, whereby either rows or columns must be on the "inside" of a table, undermined viscosity.

It is notable that all the advantages of the lish could be traced in some way to better capture of structure, and all the disadvantages to where capture of structure is still not quite sufficient. So the case for an alternative data model to the grid is strong; the lish *almost* is that model, but stumbles in a few areas.

I therefore propose (being properly wary of second system syndrome) that the next version be almost, but not quite, as simple as the lish in

its current form. I will cautiously sacrifice a little abstraction gradient to enhance capture of those structures that the lish can't quite manage, in particular by admitting coordination as a first class property. The result, I believe, will be the grid replacement that we seek.

Bibliography

R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez, and J. Saraiva. Smelling faults in spreadsheets. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 111–120. IEEE, 2014.

J. Anglim. Grammar of tables. Online: `http://jeromyanglim.blogspot.co.uk/2009/09/grammar-of-tables-tables-for-results.html`, 2009.

M. Aufreiter, A. Pawlik, and N. Bentley. Stencila – an office suite for reproducible research. Online: `https://elifesciences.org/labs/c496b8bb/stencila-an-office-suite-for-reproducible-research`, 2018.

J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, et al. The FORTRAN automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198. ACM, 1957.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Commun. ACM*, 6(1):1–17, Jan. 1963. ISSN 0001-0782. doi: 10.1145/366193.366201.

X. Badosa. The JSON-stat format. Online: `https://json-stat.org/`, 2015.

E. Bakke and D. R. Karger. Expressive query construction through direct manipulation of nested relational results. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1377–1392. ACM, 2016.

P. Bažant and M. Maršálková. A non-tabular spreadsheet with broad applicability. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 161–165, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5513-1. URL `http://doi.acm.org/10.1145/3191697.3214343`.

O. Ben-Kiki and C. Evans. YAML Ain't Markup Language (YAML) Version 1.2, 3rd Edition. Online: `http://www.yaml.org/spec/1.2/spec.html`, 2009.

A. F. Blackwell and T. R. G. Green. A cognitive dimensions questionnaire optimised for users. In *Proceedings of the Psychology of Programming Interest Group (PPIG)*, 2000.

A. A. Bock. A literature review of spreadsheet technology. IT University Technical Report Series TR-2016-199, IT University of Copenhagen, 2016. URL `https://core.ac.uk/download/pdf/81666647.pdf`.

D. Bricklin. Software Arts and VisiCalc. Online: `http://danbricklin.com/history/saincc.htm`, 2009.

M. Burnett, J. Atwood, R. W. Djang, J. Reichwein, H. Gottfried, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(02):155–206, 2001.

I. Cervesato. Nexcel, a deductive spreadsheet. *The Knowledge Engineering Review*, 22(03):221–236, 2007.

J. M. Chambers. *Programming with data: A guide to the S language*. Springer Science & Business Media, 1998.

K. S.-P. Chang and B. A. Myers. Using and exploring hierarchical data in spreadsheets. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 2497–2507. ACM, 2016.

R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 341–354, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. URL `http://doi.acm.org/10.1145/2908080.2908103`.

C. Clack and L. Braine. Object-oriented functional spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming*, 1997.

A. Clark and J. L. Hellerstein. SciSheets: Providing the power of programming with the simplicity of spreadsheets. In *Proceedings of the 16th Python in Science Conference (SciPy 2017)*. SciPy.org, 2017.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. URL `http://doi.acm.org/10.1145/362384.362685`.

D. Conway and C. Ragsdale. Modeling optimization problems in the unstructured world of spreadsheets. *Omega, International Journal of Management Science*, 25(3):313–322, 1997.

M. Csernoch and P. Biró. Problem solving in Sprego. In *Proceedings of the European Spreadsheet Risks Interests Group (EuSpRIG)*. European Spreadsheet Risks Interest Group, 2015.

M. de Vos, J. Wielemaker, H. Rijgersberg, G. Schreiber, B. Wielinga, and J. Top. Combining information on structure and content to automatically annotate natural science spreadsheets. *International Journal of Human-Computer Studies*, 103:63–76, 2017.

R. DeLine and K. Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 2*, pages 207–210. ACM, 2010.

W. Dou, C. Xu, S.-C. Cheung, and J. Wei. CACheck: Detecting and repairing cell arrays in spreadsheets. *IEEE Transactions on Software Engineering*, 43(3): 226–251, 2016.

ECMA. Standard ECMA-404: The JSON Data Interchange Format. Online: `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`, 2013.

G. Engels and M. Erwig. ClassSheets: Automatic generation of spreadsheet applications from object-oriented specifications. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 124–133, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101929.

M. Erwig. Software engineering for spreadsheets. *IEEE Software*, 26(5):25, 2009.

M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *Proceedings of the 27th International Conference on Software Engineering*, pages 136–145. ACM, 2005.

EuSpRIG. EuSpRIG horror stories. Online: `http://www.eusprig.org/horror-stories.htm`, 2019.

S. Fear. *Publication quality tables in LaTeX*, 2016. URL `http://mirror.ox.ac.uk/sites/ctan.org/macros/latex/contrib/booktabs/booktabs.pdf`.

M. Fowler. *Refactoring: Improving the Design of Existing Code*, chapter 3: Bad smells in code. Addison-Wesley, first edition, 1999.

G. W. French. The Larch environment – Python programs as visual, interactive literature. Master's thesis, School of Computing Science, University of East Anglia, 2013.

J. Friberg. Methods and traditions of Babylonian mathematics: Plimpton 322, Pythagorean triples, and the Babylonian triangle parameter equations. *Historia Mathematica*, 8(3):277–318, 1981.

R. Furuta. An integrated, but not exact-representation, editor/formatter. In J. C. van Vliet, editor, *Text Processing and Document Manipulation: Proceedings of*

the International Conference, University of Nottingham, 14-16 April 1986, pages 246–259. Cambridge University Press, 1986.

C. Granger. Light Table – a new IDE concept. Online: `http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept/`, 2012.

T. R. Green, M. M. Burnett, A. J. Ko, K. J. Rothermel, C. R. Cook, and J. Schonfeld. Using the cognitive walkthrough to improve the design of a visual programming experiment. In *Proceedings of the 2000 IEEE International Symposium on Visual Languages*, pages 172–179. IEEE, 2000.

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

T. A. Grossman and O. Ozluk. Spreadsheets grow up: Three spreadsheet engineering methodologies for large financial planning models. In *Proceedings of the European Spreadsheet Risks Interests Group (EuSpRIG)*. European Spreadsheet Risks Interest Group, 2001.

P. Guo. First impressions of the IPython Notebook. Online: `http://pgbovine.net/ipython-notebook-first-impressions.htm`, 2013.

G. Guthrie and S. McCrory. Beyond the desktop spreadsheet. In *Proceedings of the European Spreadsheet Risks Interests Group (EuSpRIG)*. European Spreadsheet Risks Interest Group, 2011.

J. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

T. Hawkins, A. Lemon, and A. Gibson. Introducing Morphit, a new type of spreadsheet technology. In *Proceedings of the EuSpRIG 2013 Conference "Spreadsheet Risk Management"*. EuSpRIG, 2013.

M. Henrion. What's wrong with spreadsheets? – and how to fix them with Analytica. Technical report, Lumina Decision Systems, 2004. URL `http://ch.lumina.com/uploads/technology/Whats%20wrong%20with%20spreadsheets.pdf`.

F. Hermans and T. Van Der Storm. Copy-paste tracking: Fixing spreadsheets without breaking them. In *Proceedings of the 1st International Conference on Live Coding (ICLC)*. ICSRiM, University of Leeds, 2015.

F. Hermans and T. van der Storm. TrueGrid: Code the table, tabulate the data. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 388–393, 2016.

F. Hermans, M. Pinzger, and A. Van Deursen. Automatically extracting class diagrams from spreadsheets. In *European Conference on Object-Oriented Programming*, pages 52–75. Association Internationale pour les Technologies Objets, 2010.

F. Hermans, M. Pinzger, and A. Van Deursen. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 451–460. ACM, 2011.

F. Hermans, M. Pinzger, and A. van Deursen. Detecting code smells in spreadsheet formulas. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 409–418. IEEE, 2012.

M. Hlavac. TableMaker: An Excel macro for publication-quality tables. *Journal of Statistical Software*, 70(3), 2016.

K. Hodnigg and M. Pinzger. xViZiT: Visualizing cognitive units in spreadsheets. In *Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on*, pages 210–214. IEEE, 2015.

R. Holwerda. Block-based languages for professionals. In *Proceedings of the Psychology of Programming Interest Group (PPIG)*, 2017.

C. D. Hundhausen and J. L. Brown. What you see is what you code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing*, 18(1):22–47, 2007.

J. Ireson-Paine. Excelsior: bringing the benefits of modularisation to Excel. In *Proceedings of the European Spreadsheet Risks Interests Group (EuSpRIG)*. EuSpRIG, 2005.

T. Isakowitz, S. Schocken, and H. C. Lucas Jr. Toward a logical/physical theory of spreadsheet modeling. *ACM Transactions on Information Systems (TOIS)*, 13 (1):1–37, 1995.

B. Jansen and F. Hermans. Using a visual language to create better spreadsheets. In *Proceedings of the First Workshop on Software Engineering Methods in Spreadsheets*, pages 48–51, 2014.

E. Kandogan, J. Kim, T. P. Moran, and P. Pedemonte. How a freeform spatial interface supports simple problem solving tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 925–934. ACM, 2011.

M. B. Kery. Tools to support exploratory programming with data. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 321–322. IEEE, 2017.

D. Kirsh. The intelligent use of space. *Artificial Intelligence*, 73(1):31–68, 1995.

A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, et al. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):21, 2011.

T. Y. Lee, C. Dugan, and B. B. Bederson. Towards understanding human mistakes of programming by example: an online user study. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces*, pages 257–261. ACM, 2017.

L. Leon, D. M. Abraham, and L. Kalbers. Beyond regulatory compliance for spreadsheet controls: A tutorial to assist practitioners and a call for research. *Communications of the Association for Information Systems*, 27(28):541–560, 2010.

L. Leon, Z. H. Przasnyski, and K. C. Seal. Introducing a taxonomy for classifying qualitative spreadsheet errors. *Journal of Organizational and End User Computing (JOEUC)*, 27(1):33–56, 2015.

H. Lieberman, F. Paternò, M. Klann, and V. Wulf. *End-User Development: An Emerging Paradigm*. Springer, 2006.

B. Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *2009 IEEE 25th International Conference on Data Engineering*, pages 417–428. IEEE, 2009.

N. Macedo, H. Pacheco, N. R. Sousa, and A. Cunha. Bidirectional spreadsheet formulas. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–168. IEEE, 2014.

N. Mangano, H. Ossher, I. Simmonds, M. Callery, M. Desmond, and S. Krasikov. Blending freeform and managed information in tables. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 840–843. IEEE, 2011.

R. M. McCutchen, S. Itzhaky, and D. Jackson. Object Spreadsheets: a new computational model for end-user development of data-centric web applications. In *Onward! 2016: Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 112–127, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4076-2.

W. McKinney and PyData Development Team. Pandas: powerful Python data analysis toolkit. Online: `http://pandas.pydata.org/pandas-docs/stable/pandas.pdf`, 2014.

A. McNamara. Key attributes of a modern statistical computing tool. *The American Statistician*, 2018.

J. Mendes and J. Saraiva. Tabula: A language to model spreadsheet tables. In *Proceedings of the International Symposium on End-User Development*, 2017.

G. Miller and F. Hermans. Gradual structuring in the spreadsheet paradigm. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 240–241. IEEE, 2016.

R. Mittermeir and M. Clermont. Finding high-level structures in spreadsheet programs. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 221–232. IEEE, 2002.

J. Mount and N. Zumel. Coordinatized data: A fluid data specification. Online: `http://winvector.github.io/FluidData/RowsAndColumns.html`, 2017.

R. Muenchen. The popularity of data analysis software. Online: `http://r4stats.com/articles/popularity/`, 2015.

T. E. Oliphant et al. NumPy v1.9 Manual. Online: `http://docs.scipy.org/doc/numpy/user/basics.rec.html`, 2014.

J. F. Pane and B. A. Myers. More natural programming languages and environments. In H. Lieberman, F. Paternò, and V. Wulf, editors, *End User Development*, pages 31–50. Springer, 2006.

R. R. Panko. What we don't know about spreadsheet errors today: The facts, why we don't believe them, and what we need to do. In *Proceedings of the EuSpRIG 2015 Conference "Spreadsheet Risk Management"*. European Spreadsheet Risks Interest Group, 2015.

R. R. Panko and S. Aurigemma. Revising the Panko–Halverson taxonomy of spreadsheet errors. *Decision Support Systems*, 49(2):235–244, 2010.

S. Peyton Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in Excel. *ACM SIGPLAN Notices*, 38(9):165–176, 2003.

P. G. Polson, C. Lewis, J. Rieman, and C. Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36(5):741–773, 1992.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. URL `https://www.R-project.org/`.

R Project for Statistical Computing. *An Introduction to R*, 2019. URL `https://cran.r-project.org/doc/manuals/r-devel/R-intro.html#Vector-arithmetic`. Section 2.2.

J. F. Raffensperger. New guidelines for spreadsheets. In *Proceedings of the European Spreadsheet Risks Interests Group (EuSpRIG)*. European Spreadsheet Risks Interest Group, 2001.

C. Roast, D. Patterson, and V. Hardman. Visualisation – it is not the data, it is what you do with it. In *Proceedings of e-Society 2018, 16th International Conference*, page 231. International Association for Development of the Information Society, 2018.

B. G. Ryder, M. L. Soffa, and M. Burnett. The impact of software engineering research on modern programming languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(4):431–477, 2005.

M. V. Sadaphule and N. F. Shaikh. SQL Query Parser: An automated tool for translating the queries into spreadsheets. *International Journal of Computer Science and Information Security*, 14(8):23, 2016.

A. Sarkar, A. D. Gordon, S. Peyton Jones, and N. Toronto. Calculation view: multiple-representation editing in spreadsheets. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 85–93. IEEE, 2018.

C. Scaffidi. The impact of human-centric design on the adoption of information systems: A case study of the spreadsheet. In *2016 11th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, 2016.

P. Sestoft. Spreadsheet technology. IT University Technical Report Series TR-2011-142, IT University of Copenhagen, 2012. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.362.4106&rep=rep1&type=pdf`.

P. Sestoft. Online partial evaluation of sheet-defined functions. *Electronic Proceedings in Theoretical Computer Science*, 129(Festschrift for Dave Schmidt): 136–160, September 2013.

J. Tandy and I. Herman. Generating JSON from Tabular Data on the Web. W3C Recommendation, W3C, 2015.

S. L. Tanimoto. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*, pages 31–34. IEEE, 2013.

D. Thomas and A. Hunt. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

J. Tyszkiewicz. Spreadsheet as a relational database engine. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 195–206. ACM, 2010.

A. Unwin. Oscars and interfaces. *Journal of Statistical Software*, 49(11), 2012.

P. M. Valero-Mora and R. Ledesma. Graphical user interfaces for R. *Journal of Statistical Software*, 49(1), 2012.

B. Victor. Learnable programming. Online: `http://worrydream.com/LearnableProgramming/`, 2012.

X. Wang. *Tabular abstraction, editing, and formatting*. PhD thesis, Department of Computer Science, University of Waterloo, Ontario, 1996.

H. Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21(12), 2007.

H. Wickham. Tidy data. *Journal of Statistical Software*, 59(10), 2014.

Appendix A

Interviewer's script for user study

A new model for spreadsheet-like tabular data:
Questionnaire and Interviewer Script

Alan Hall, The Open University
alan.hall@open.ac.uk

Participants will already have seen a presentation introducing the "lish" and the prototype editor.

Thank you for taking part in this study on **representations for tabular data**. It is in the form of an interview where some questions are multiple choice and some open-ended. I will also be asking you to carry out a task using a prototype table editor. You can pass on any question if you don't know or prefer not to answer.

Q1. The first question is about **spatial layout** in tables. Please take a look at these tables, which show some data on rainfall at two locations broken down by month and year. *Show Cards A & B; make zoomed version visible on screen too, in case participant finds print rather small.* There are three versions, all showing the same data but in different formats. Thinking about the sorts of analysis you might want to do with the data, or summary statistics you might want to produce, could you tell me what you would consider to be the pros and cons of the three formats?

Allow participant to discuss. Follow up as appropriate:

Here are a few [more] to think about.

- Which one conveys the clearest visual impression of the scope and coverage of the data?
- Which one would you use (by eye) to find quickly the wettest April in each location?
- Would you chose differently if wanting to do that by code, perhaps in a larger dataset covering more locations and years?
- Would your choices depend on which tool you were using (e.g. manual calculation, spreadsheet, R code, ...)?
- How about obtaining total annual rainfall, by location and year?

Warm up task

For the next part of the session, we will be working with the "lish" data representation, using a prototype lish editor. First I will take you through the basic operation of the editor. Then I will ask you to carry out a task using it. After that I will have some questions about the task, and about using the lish for modelling more generally.

Take participant through warm up task here.

Main task

Now I'm going to ask you to build a model on your own. I will give you the model as a spreadsheet and ask you to replicate it as a lish. Please work on your own as much as possible, but do ask for help if you get stuck.

Participant to try doing the main task here. To include an extension of their choice to the specified model.

Record whether participant successfully built the model. At what points, if any, was intervention needed?

Questions on the main task

The next questions are about how easy or hard you found various aspects of building the model. In the multiple choice questions, feel free to "think aloud" when arriving at your choice, and optionally to add verbal comments to your chosen answer.

Thinking about **setting up the initial structure**, before you added any calculations, please score each of the following statements using this list. *Show Card C [7 point scale from Strongly Disagree to Strongly Agree]*. After the first few, I will check back in case you want to recalibrate where your answers lie on the scale.

Q2. I could easily see how to create and populate cells and lists.

Q3. I understood the "template" behaviour of the first item in my lists, and what patterns were going to repeat as a result.

Q4. From the available building blocks of cells and lists, it was clear how to structure my model.

Q5. When viewing and navigating the model, I could tell what the underlying list structure was.

Q6. When I wanted to insert the extra level of [some value], I could easily locate the cursor appropriately for the insert operation.

Now that you have seen a few of the questions, can I just check whether you want to recalibrate where your answers lie on the scale? Feel free to revise any answers if appropriate. *Review answers to Q2-Q6 if participant would like.*

And now some questions about the **calculations**.

Q7. I could easily locate where to put my formulae so that the correct **output cells** were selected.

Q8. While building the formulae, I could easily find the right location such that the correct **input cells** were selected.

Q9. When performing "vectorised" calculations, I understood the relationship between the formula and the contents of the individual cells.

Q10. When I specified a "sum" operation, the machine interpreted it the way I expected from the shape of the layout.

And a couple of more open questions (not multiple choice) about the prototype editor.

Q11. Was the behaviour of the system **surprising** to you at any point while building your model?

Q12. Were there any parts of the model building process that felt **unwieldy** or didn't "flow"?

Questions relating the lish to participant's own analysis

In the last block of questions, I am interested in relating the lish to the data structures you encounter in your job as a professional analyst. I am only interested in the structural patterns, so I will not be asking you about the business context. When I ask about your business as usual (BAU), I mean any work that you have been directly involved with in a current or previous role – either because you did the analysis yourself, or supervised somebody who did it.

Q13. Approximately how long have you worked as an analyst (including both in GORS and any previous jobs involving significant analysis)?

Q14. Please think about some example data structures you've encountered in your BAU, casting your mind over some models you've worked on, now or in the past. Could you please comment on how prevalent lish-like structures are these models? As a reminder, we've seen that the lish represents:

- hierarchical structures – which you might think of as “boxes within boxes”;
- repeating patterns – such as many tables of a similar structure or requiring the same calculation;
- systems where some dimensions are fixed, and others allowed to vary.

Q15. The next question is about trade-offs: using the lish might make some things easier, but others harder, compared to a spreadsheet, R script or some other type of tool you may be using. A lish model might mitigate some risks of errors, but aggravate others. Can you think of any such trade-offs that would apply to your own work?

Let participant respond. Then prompt if not already covered:
Here are a few [more] to think about:

- Initial lish creation: upfront effort vs. later payback.
- Cells and lists: simplicity of building blocks vs. needing to assemble higher level structures yourself.
- Live updates: consistency and convenience vs. reduced transparency.
- Vectorised formulae: DRYness vs. conceptual difficulty.

Now I'd like to recall three main ways in which the lish tries to support the user during the modelling process. I will ask you to score them on this grid, and discuss why you gave your scores. *Show Card D.* You can see that the grid comprises two dimensions: whether the lish supports each aspect of workflow effectively, and whether that aspect is important to your own BAU.

The three aspects are:

Q16. The lish maintains consistency of structures, with live updates.

Q17. The lish allows formulae to be fewer and simpler (compared either to a spreadsheet, or to expressions you might have to compose in a script).

Q18. The lish provides an interactive way to refer (for example) to multiple columns, from separate but related tables.

Open comments

Q19. Finally, is there anything else you would like to comment on?

CARD A. RAINFALL DATA (1 of 2) – Monthly totals in millimetres – Long form (1)

location	year	month	rain
Eastbourne	2014	Jan	163
Eastbourne	2014	Feb	124
Eastbourne	2014	Mar	48
Eastbourne	2014	Apr	44
Eastbourne	2014	May	60
Eastbourne	2014	Jun	10
Eastbourne	2014	Jul	63
Eastbourne	2014	Aug	100
Eastbourne	2014	Sep	10
Eastbourne	2014	Oct	155
Eastbourne	2014	Nov	157
Eastbourne	2014	Dec	58
Eastbourne	2015	Jan	129
Eastbourne	2015	Feb	65
Eastbourne	2015	Mar	25
Eastbourne	2015	Apr	8
Eastbourne	2015	May	71
Eastbourne	2015	Jun	21
Eastbourne	2015	Jul	62
Eastbourne	2015	Aug	148
Eastbourne	2015	Sep	103
Eastbourne	2015	Oct	38
Eastbourne	2015	Nov	102
Eastbourne	2015	Dec	86
Eastbourne	2016	Jan	168
Eastbourne	2016	Feb	54
Eastbourne	2016	Mar	45
Eastbourne	2016	Apr	26
Eastbourne	2016	May	44
Eastbourne	2016	Jun	75
Eastbourne	2016	Jul	5
Eastbourne	2016	Aug	22
Eastbourne	2016	Sep	22
Eastbourne	2016	Oct	33
Eastbourne	2016	Nov	70
Eastbourne	2016	Dec	12
Stornoway	2014	Jan	139
Stornoway	2014	Feb	148
Stornoway	2014	Mar	105
Stornoway	2014	Apr	86
Stornoway	2014	May	63
Stornoway	2014	Jun	49
Stornoway	2014	Jul	66
Stornoway	2014	Aug	141
Stornoway	2014	Sep	48
Stornoway	2014	Oct	198
Stornoway	2014	Nov	90
Stornoway	2014	Dec	168
Stornoway	2015	Jan	199
Stornoway	2015	Feb	115
Stornoway	2015	Mar	167
Stornoway	2015	Apr	65
Stornoway	2015	May	121
Stornoway	2015	Jun	99
Stornoway	2015	Jul	110
Stornoway	2015	Aug	89
Stornoway	2015	Sep	37
Stornoway	2015	Oct	71
Stornoway	2015	Nov	156
Stornoway	2015	Dec	268
Stornoway	2016	Jan	143
Stornoway	2016	Feb	146
Stornoway	2016	Mar	86
Stornoway	2016	Apr	55
Stornoway	2016	May	63
Stornoway	2016	Jun	83
Stornoway	2016	Jul	143
Stornoway	2016	Aug	92
Stornoway	2016	Sep	141
Stornoway	2016	Oct	38
Stornoway	2016	Nov	98
Stornoway	2016	Dec	112

CARD B. RAINFALL DATA (2 of 2)

Monthly totals in millimetres

Wide form (2)

Eastbourne	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2014	163	124	48	44	60	10	63	100	10	155	157	58
2015	129	65	25	8	71	21	62	148	103	38	102	86
2016	168	54	45	26	44	75	5	22	22	33	70	12

Stornoway	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2014	139	148	105	86	63	49	66	141	48	198	90	168
2015	199	115	167	65	121	99	110	89	37	71	156	268
2016	143	146	86	55	63	83	143	92	141	38	98	112

Alternative wide form (3)

2014	Eastbourne	Stornoway
Jan	163	139
Feb	124	148
Mar	48	105
Apr	44	86
May	60	63
Jun	10	49
Jul	63	66
Aug	100	141
Sep	10	48
Oct	155	198
Nov	157	90
Dec	58	168

2015	Eastbourne	Stornoway
Jan	129	199
Feb	65	115
Mar	25	167
Apr	8	65
May	71	121
Jun	21	99
Jul	62	110
Aug	148	89
Sep	103	37
Oct	38	71
Nov	102	156
Dec	86	268

2016	Eastbourne	Stornoway
Jan	168	143
Feb	54	146
Mar	45	86
Apr	26	55
May	44	63
Jun	75	83
Jul	5	143
Aug	22	92
Sep	22	141
Oct	33	38
Nov	70	98
Dec	12	112

CARD C. SEVEN-POINT ANSWER SCALE

7	Strongly agree
6	Agree
5	Tend to agree
4	Neutral
3	Tend to disagree
2	Disagree
1	Strongly disagree

0	Don't know / Not applicable / Other

CARD D. TWO DIMENSIONAL ANSWER GRID

The lish supports this aspect of workflow effectively.								
	7 Strongly agree	Wrong problem, right solution					Right problem, Right solution	
	6 Agree							
	5 Tend to agree							
	4 Neutral				Neutral			
	3 Tend to disagree							
	2 Disagree							
	1 Strongly disagree	Nul points					Right problem, wrong solution	
		1 Strongly disagree	2 Disagree	3 Tend to disagree	4 Neutral	5 Tend to agree	6 Agree	7 Strongly agree
		This aspect of workflow is important to my BAU						

0	Don't know / Not relevant / Other

Appendix B

Editor "crib sheet" for user study

Participants were provided with a copy of the two pages that follow to refer to when attempting the task.

Lish Editor crib sheet

Switching windows

Enter	(on blank line, in console window)	Switch to viewer window
ESC	(in viewer window)	Switch to console window
ESC	(in console window)	Cancel command being edited

Or you can simply click in the required window

Forming and editing lishes (in viewer window)

Spacebar	Insert element (cell or sublist) at cursor
Control-K	(**K**ill) Delete element at cursor
a-z, 0-9	Initiate command to enter number or string in a cell If you are in a margin, this creates a **label** If you are in a list body, this creates a **value**
F2	Edit existing cell contents – label or value, as above
Shift-F2	Edit existing cell **value** (always the value, even if cell is in a margin!)
Delete	Clear all values, labels and formulae from selection. The structure is left in place.

Note you can't edit the outermost marginal cells (gives message about a “Thick View”)

Sublists

[	(open square bracket)	Enclose cursor item into a sublist, with blank initial cell
]	(close square bracket)	Reverse the above, dissolving the sublist into its parent
,	(comma)	Pull next item into selected sublist
.	(period)	Push last item out of selected sublist
Control-O		(**O**rientation) Cycle selected sublist through possible orientations

Navigation

Arrow keys	Move cursor
Enter	Drill into a sublist by one level. If already at cell level, advance by one cell.
Backspace	Drill out to next higher level
Control-R	(**R**ectangles) Turn frames of parent sublists on or off.
Control-Home	Go to top left cell
Control-G	**G**o to cell (specified by a string containing its name or id, with no dollar)

Creating tables / arrays

Press [to create a sublist, and Enter to drill into it.
Press [again to enclose the first cell of the sublist, and Enter to drill in again.
You now have a 2 x 2 table (including margins). Repeat the sequence if you want higher dimensions.
Press Spacebar at the appropriate levels to add further rows and columns.
Press [within a margin to create a **row or column group**.

Creating families of “objects”

Create the first member of the family using any of the commands above.
Use [and , as required, to make this first member reside in a **single sublist**.
Press Control-L (c**L**one) to form the family based on this sublist (Control-M to revert).
Press Spacebar to create further family members.

Formulae (must not contain forward references)

=	(Equals)	in the viewer window: initiate command to enter formula in current cell
$	(Dollar)	in the **console** window: switch back to viewer so you can navigate to a range you want to include in the formula
$	(Dollar)	in the **viewer** window: insert name of current cell / range in the console window. If it has no name, a unique numeric identifier is generated.
F9		Recalculate all formulae. This is done automatically when you edit a formula, but not when you edit a value.

Named ranges in formulae are prefixed by a dollar sign, e.g. `$myrange`

Arithmetic operators (note will produce CalcErr if there are missing values)

+ - / *	Vectorised arithmetic operators
< <= == != >= >	Vectorised comparison operators (use `bool2bin` if displaying result)
& |	Vectorised logical operators (AND, OR respectively)
- !	Vectorised unary operators (minus, NOT)

Functions

`sum(expr)`	
`count(expr)`	
`mean(expr)`	
`min(expr)`	
`max(expr)`	
`paired_min(expr1, expr2)`	Like `pmin` in R
`paired_max(expr1, expr2)`	Like `pmax` in R
`abs(expr)`	Absolute value
`sqrt(expr)`	Square root
`bool2bin(condition)`	Convert a boolean (true/false) vector into binary (1/0)
`ifelse(condition, result_when_true, result_when_false)`	
`filter(range, condition)`	NOTE: results of `filter` are undefined for missing values!
`seq(start, step, length)`	Simple integer sequences
`match(what, where)`	Return indices of elements of `what` in the vector `where`
`index(range, indices, depth)`	Return elements of range by position (indices) and depth
`lookup(what, where, range, depth)`	Combines `match` and `index` in a single function

Examples

```
$price * $vatrate
$var1 + $var2 + 100
sum($daily_consumption)
paired_max($myvalue, 0)  # replaces negative values of $myvalue with 0
ifelse($delay > 5, "delayed", "on time")
filter($item, $unit_cost > 50)
```

To exit

Press F12, or type “exit” at the console. WARNING: does not prompt you for unsaved work!