Object Spreadsheets: an end-user development tool for web applications backed by entity-relationship data

by

Richard Matthew McCutchen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2016

Certified by .....
Daniel Jackson
Professor
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Object Spreadsheets: an end-user development tool for web applications backed by entity-relationship data

by
Richard Matthew McCutchen

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

There is a growing demand for *data-driven web applications* that help automate organizational and business processes of low to medium complexity by letting users view and update structured data in controlled ways. We present Object Spreadsheets, an end-user development tool that combines a spreadsheet interface with a rich data model to help the process administrators build the logic for such applications themselves. Its all-in-one interface with immediate feedback has the potential to bring more complex tasks within reach of end-user developers, compared to existing approaches.

Our data model is based on the structure of entity-relationship models and directly supports nested variable-size collections and object references, which are common in web applications but poorly accommodated by traditional spreadsheets. Object Spreadsheets has a formula language suited to the data model and supports stored procedures to specify the forms of updates that application users may make. Formulas can be used to assemble data in the exact structure in which it is to be shown in the application UI, simplifying the task of UI building; we intend for Object Spreadsheets to be integrated with a UI builder to provide a complete solution for application development.

We describe our prototype implementation and several example applications we built to demonstrate the applicability of the tool.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

I'd like to thank my advisor Daniel Jackson for giving me the opportunity to do work I care deeply about and the guidance to make it a success, and my teammate Shachar Itzhaky for driving the design, implementation, and writing forward when I was stuck seeking perfection in some limited domain, without which the project never would have gotten this far. Much of the text of this thesis has been adapted from our technical report, available at `http://dspace.mit.edu/handle/1721.1/100803`, with some revisions; however, almost all of Sections 3.2 and 3.3 and Chapter 7 is new for the thesis.

This project was supported by a grant from Wistron Corporation as part of a collaboration between Wistron and MIT's Computer Science and Artificial Intelligence Laboratory. This project also received partial support from the National Science Foundation under Grant No. CCF-1438982, "XPS: FULL: FP: Collaborative Research: Model-based, Event Driven Scalable Programming for the Mobile Cloud".

We would like to thank Edward Doong for his help developing example applications; David Karger and Eirik Bakke for their advice on the design of the system, positioning our work and the user study; Jonathan Edwards for early discussions; and various other colleagues at MIT and the anonymous reviewers who provided feedback on our work and drafts of our papers. We are grateful to Wistron Corporation and the National Science Foundation for their support of this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

A wide class of *data-driven web applications* involve routine but non-trivial manipulations of data, to support sharing of information, small-scale social interactions, and business processes. For example, a school arranging parent-teacher conferences may want an application that lets the family of each student schedule a meeting with each of the student's teachers, avoiding problems such as double-booking (Fig. 1-1).

Organizations with such a need face a dilemma: to adopt an off-the-shelf solution (which may be a less than perfect match to the requirements); to engage a developer (which is usually too expensive); or, as is most commonly done, to cobble together tools such as email, spreadsheets and online forms using form builders like Google Forms [3] and Wufoo [6] (leaving a considerable burden of manual work and possibly undesirable risks to data confidentiality and integrity).

Ideally, an organization's administrators would build a application themselves to their exact requirements (*end-user development*), but this approach is not as easy or as widely used as it could be. General-purpose web application frameworks continue to demand a high level of technical understanding from their users, even as design advances over time, such as scaffolding scripts and object-relational mapping, reduce the amount of code that has to be written. Existing *application builders* (such as

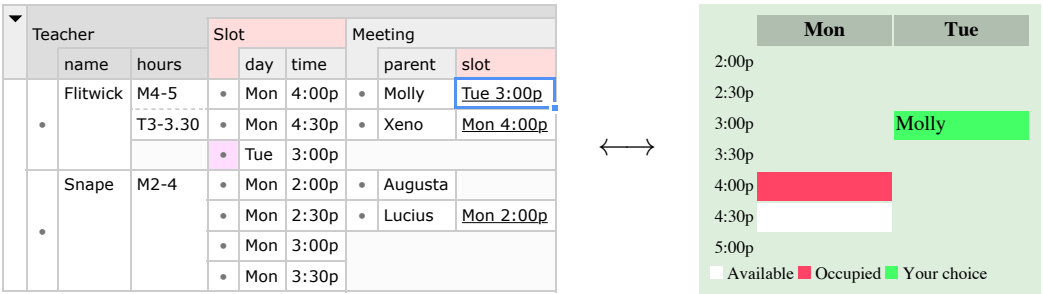


Figure 1-1: A simple parent-teacher conference application, with the spreadsheet on the left and a web UI on the right.

App2You [27] and Intuit QuickBase [1]) make it easier for people with little or no programming experience to build data-driven applications of low to medium complexity, offering menu-driven, WYSIWYG, or other visual interfaces to specify the structure of the information stored and the ways in which users may view and update it. Such tools are more general than form builders, which allow unprivileged users to add records but offer very limited options (if any) for them to view and edit records other than their own.

Many organizations use application builders to great effect, but others continue to use piecemeal solutions, likely because they are intimidated even by the apparent investment needed to learn an application builder. And advanced developers often prefer to use general-purpose frameworks if they have the expertise, even though that can make the applications harder for new people to maintain later; we suspect this is because they assume that an application builder will not give them the level of power and control they want.

We propose that the use of the spreadsheet paradigm in an application builder can help to overcome both of these problems. In general, spreadsheets are the most successful example of an end-user development paradigm, and for this reason, are a common focus of attempts to bring greater ease of use to programming. They have been extended with capabilities such as more powerful programming languages [35, 16], user-defined functions [11, 34], and stream processing [38]. The appeal of spreadsheets arises from several aspects:

- A simple and flexible visual structure for organizing data;
- The use of the very same structure and interface not only for data, but also for the schema underlying the data and the formulas defining queries on it;
- The continual computation strategy, which allows the impact on example data to be viewed as formulas are constructed and modified;
- A simple and declarative formula language that provides a smooth learning curve from simple data transformations, which can be selected visually, to more complex transformations written with the help of integrated language documentation and immediate feedback on subexpression results.

In the context of building a data-driven web application, one can hope that the use of spreadsheets and a suitable formula language will:

- Make it easier for end-user developers to progressively construct the schema they need to fit their data via the single spreadsheet interface, compared to existing application builders in which developers have to go back and forth between different screens.
- Provide a smooth path for developers who are comfortable designing a spreadsheet to hold their data to start building an application around that spreadsheet, a leap that otherwise might have been intimidating.

- Provide a smooth learning curve for developers to express simple to moderately complex application logic.

- Lead to a programming environment that feels general enough to win the respect of advanced developers, even as it offers significant guidance to novice developers.

However, adapting the spreadsheet paradigm to the development of data-driven web applications is not trivial, and in Section 1.4 we outline particular challenges. In fact, Quilt [10] is an example of a web application builder backed by a traditional spreadsheet in the style we propose, but it does not address most of these challenges and consequently supports only the very simplest applications. Our aim in this project, therefore, has been to retain the essential appeal of the spreadsheet paradigm while providing good solutions to these challenges.

1.2 Overview

In this thesis, we propose an enhanced spreadsheet tool, called Object Spreadsheets, that can be used by an end-user developer to build all of the logic for a data-driven web application. The spreadsheet serves as the standard UI for development and administrative access to the application data; the developer would design a separate, customized *application UI* for regular use by unprivileged users. (Throughout this thesis, we consistently use the term "developer" for a person who defines the schema and logic of an application or spreadsheet, however simple, and "user" for a person who merely reads and writes data.)

In addition to the editable sheet with formulas, Object Spreadsheets supports stored procedures to define the updates that users of an application can make, and exposes an API for the application UI to display data and invoke procedures. We envision combining the tool with a suitable UI builder to provide a complete solution for application development that requires no prior knowledge of web technologies. While some parts of the development process (particularly schema design) will remain challenging for many end-user developers due to the level of abstraction involved, and form building approaches will remain helpful, we believe that offering a spreadsheet interface has the potential to bring a range of development tasks within the reach of end-user developers for a range of realistic target applications and scenarios.

We have implemented a prototype of the spreadsheet tool in the Meteor web application framework and demonstrated it on a collection of example applications, building the application UIs directly in Meteor using the exposed API. We have also conducted a user study to collect feedback from developers who might use such a tool.

The rest of the thesis describes in more detail how the logic of a data-driven web application can be built using Object Spreadsheets. Our contributions include:

- An analysis of the challenges of extending the spreadsheet paradigm to support web application development (Section 1.4);

- A data model (Chapter 2) and spreadsheet interface designed to handle web application data in a way that is natural to end-user developers;
- A simple formula language that supports relational queries (Chapter 3), and a simple procedural language to support restricted updates (Chapter 4);
- A prototype implementation of the tool (Section 5.1);
- A suite of example applications that demonstrate common difficulties presented by this application class, and their implementations in our tool (Section 5.2);
- Qualitative feedback from a user study (Section 5.4).

1.3 Demos

Interactive demos of the example applications and a video demonstrating how to build an application with Object Spreadsheets are available on the project web site at `http://sdg.csail.mit.edu/projects/objsheets/`. These materials may help the reader quickly get a sense of what Object Spreadsheets does before reading further.

	A	B	C	D	E	F
1	room	sq footage	occupant	role	alloc	free
2	Dungeon Five	480	Sirius	Grad student	12	**436**
3			James	Post-doc	20	
4			Wormtail	Grad student	12	
5	Greenhouse Two	561	Bellatrix	Visiting Prof	45	**476**
6			Lily	Post-doc	20	
7			Remus	Post-doc	20	

8	role	alloc space	=VLOOKUP(D5, A$9:B$11, 2)
9	Grad student	12	=B2−sum(E2:E4)
10	Post-doc	20	=B5−sum(E5:E7)
11	Visiting Prof	45	

Figure 1-2: Tracking the used space in each room at a university department based on space allocation amounts for each role. Adding an occupant or room requires careful adjustment of the formulas. A real implementation would have a separate table of people like the table of roles above, but we omit this detail to simplify the example.

1.4 Challenges

In this section we outline the key challenges of extending a spreadsheet to support web application development and how they are addressed in our design. We point out a few alternatives and explain why they are inferior. Further discussion of alternatives and similar systems is given in Chapter 6.

1.4.1 Nested Variable-Size Sets

All but the simplest web applications contain one or more sets of objects and allow users to add objects to and remove objects from these sets. Many even include two or more nested levels of such sets. For example, consider an application used by an administrator to manage the space allocation for a university department, shown in spreadsheet form in Fig. 1-2. Each person is allocated an amount of space depending on their role, and people must be assigned to rooms in such a way that each room is large enough for the people assigned to it. The administrator is constantly facing the problem of finding a room with enough free space to accommodate the next person, so he wrote formulas that subtract the total allocated space from the square footage of each room. Unfortunately, this requires hard-wiring the cell ranges corresponding to the occupants of each room (E2:E4 and E5:E7). The impact of this is that when adding a new occupant to a room or when adding a room to the list, the formulas have to be edited or copied to consider the new cells.

This example illustrates the fundamental challenges of handling variable-size sets in a spreadsheet. Applications require both per-item computations, such as the lookup of the allocated space for each person based on their role, and computations over sets,

such as summing the allocated space for the occupants of a room. Actually, the latter computation is also a per-item computation at the room level. To support variable-size sets, a spreadsheet must be able to:

1. Fit as many items as are needed;
2. Automatically apply per-item computations to added items;
3. Maintain enough information to locate sets and their enclosing items as sizes change.

These capabilities are difficult to achieve in a traditional spreadsheet, in which data items and formulas are bound to individual cells in the grid and there is no paradigm for adapting the structure to programmatic changes in data size. One strategy to handle two levels of sets is to lay out one level (e.g. the rooms) along one axis, and another level (e.g. the occupants) along the other. This limits nesting to two levels, and is also hard to maintain when the inner items are composed of several fields, as in the example. Another strategy is to move all the inner items to a separate table with references to the outer items, as in a relational database. But if end-user developers think of their data as nested, this transformation is an ongoing burden and in particular risks making schema design even harder than it already is.

In Object Spreadsheets, we solve the problem by abandoning the two-dimensional grid of cells as the fundamental data model in favor of a richer model that is merely viewed in a two-dimensional layout. The model is never dependent on the spreadsheet view for its correct functioning, and the view adapts to arbitrary changes in the size of the model; there are no issues of "running out of room in the grid". Some commands in the spreadsheet UI depend on the state of the view at the time of invocation, but they ultimately result in a change to the model that is expressed in view-independent terms.

The model we choose is based on what is historically known as the "hierarchical data model with virtual records"; it directly supports nested variable-size sets of objects. Every formula defines a computed field of an object type and is automatically

Room			Occupant			
	name	sqFoot		name	role	free
	text	number		text	Role	number
•	Dungeon Five	480	•	Sirius	Grad student	436
			•	James	Post-doc	
			•	Wormtail	Grad student	
•	Greenhouse Two	561	•	Bellatrix	Visiting Prof	476
			•	Lily	Post-doc	
			•	Remus	Post-doc	

Role		
	title	allocSpace
	text	number
•	Grad student	12
•	Post-doc	20
•	Visiting Prof	45

```
Room.free ≙ sqFoot - sum[v : Occupant](v.role.allocSpace)
```

Figure 1-3: An Object Spreadsheet for the space allocation example.

evaluated on each object of that type in the sheet, ensuring uniformity. A root object is available to hold global values and formulas. The spreadsheet view uses the "nested table" layout with each object type occupying a range of columns and objects of the same type occupying vertically stacked rectangles, which is common in other tools [8, 7, 23]. The result for the space allocation example is shown in Fig. 1-3.

1.4.2 Object References

Web applications include relationships between objects, not all of which are well captured via hierarchy, which leaves a need for object references of some form. In the space allocation example (Fig. 1-2), this can be seen by the "role" of each occupant, which refers to a role listed in the table at the bottom. The administrator then wants to retrieve the allocated space for the role from this table. This can be done with the VLOOKUP function, but it becomes tedious and error-prone to specify the target cell range for each such lookup in an application. This approach is the analogue of a join or subquery on a foreign key in a relational database.

Another approach is to have the occupant's role cell store a cell reference in string form, such as "A11" in the case of Bellatrix, and use a formula like =OFFSET(INDIRECT(D5),0,1) to look up the allocated space. This approach avoids specifying the cell range of the role table but still requires significant boilerplate, and it fails if the role table must be moved to allow the room table to grow. Furthermore, to enter the role of an occupant into the sheet, the administrator has to manually look up the correct cell reference.

Object Spreadsheets provides object references that are analogous to the "A11" mentioned above, but since the data model supports objects directly, these references do not break when the layout changes. So if the developer defines an *object type* named Role corresponding to the role table, then references to individual Role objects can be stored in the "role" column of the occupant table and manipulated like any other data type. A dot notation is used to access fields of the target object, so the lookup of the allocated space might be expressed as "=role.allocSpace". The same notation is used to access ancestor or child objects in the hierarchy, for example, to retrieve all occupants of a room to compute the free space. We call these dot expressions *navigations*. Finally, Object Spreadsheets lets the developer designate one field of each object type (defaulting to the first field) as its "display" field, which is used as a string representation to display and input references to objects of that type in the spreadsheet. So, by default, references to roles from the occupant table would display the role name, underlined to remind the developer that they are references.

1.4.3 Binding the Application UI to Data

Any web application builder has to give the developer a way to specify what data should appear in a given page of the (user-facing) application UI. In existing tools such as QuickBase and App2You, each page is associated with a particular object type, and a form building interface is used to specify what fields of the object type and what tables of related objects to display. The amount of logic inherent in such

UI building becomes nontrivial if related objects are nested or are filtered, potentially depending on parameters chosen by the user on the page.

We propose to harness the benefits of spreadsheets for this task by making each page merely a stylized view of a dedicated region of the spreadsheet that contains the data for display. As described in the previous paragraphs, our spreadsheet model has a hierarchical structure, which aligns well with how user interfaces are normally built, plus plenty of expressive power to select and assemble data. We discuss further details in Section 3.4.

1.4.4 Mutations

Finally, the developer must specify the kinds of mutations that users may make to the application's state. Suppose the administrator from the space allocation example wants to allow other users to assign occupants, but not e.g. add rooms or alter their square footage. In the simplest case, if a web application builder has a flexible means to bind mutable state directly to a page, the developer could choose to allow edits to some of the displayed values and creation and deletion of objects that meet the criteria to appear in tables on the page, perhaps subject to conditions expressed as formulas. If the page is bound to a view defined by formulas on the source data, then the natural way to achieve equivalent functionality is to provide default view-update semantics for formulas with appropriate syntactic forms, as (for example) PostgreSQL does.

However, some of our target applications, such as Got Milk (see Section 5.2), have composite mutations that cannot be expressed in this form without complicated tricks. The most basic, general way to support such mutations is to use stored procedures and allow users to call certain procedures with arguments of their choice; the procedures would also receive some built-in parameters such as the identity of the calling user and the time. This is the approach we currently take. We designed a small procedural language as extension of the formula language (thus, we hope, making it easy for end-users to understand). A room assignment procedure might look like this:

```
assign  (room: Room, name: text, role: Role)
          let a = new room.Occupant
          a.name = name
          a.role = role
          check room.free >= 0
```

Chapter 2

Data Model

2.1 Features

The data model for a sheet in Object Spreadsheets is based on what is historically known as the "hierarchical data model with virtual records", since we believe this model offers the best compromise between closeness to an end-user developer's mental model of an application (which we imagine to be an entity-relationship model) and practicality of visualization. The essential features of this model are:

- Attributes, or *fields*, as the basic unit of data (rather than relational tuples)
- Abstract references between objects
- An ownership hierarchy of objects

Our model also shares some features with the functional data model of DAPLEX [36] (though we currently do not offer analogues of many of its more advanced features):

- Set-valued fields
- Definition of a computed field on an object type by a formula, which is automatically evaluated once for each object of the type

(There are further similarities to DAPLEX in the formula and procedural languages, as we will see in the respective chapters.) The two-dimensional grid presentation of the data model approximates the "nested table layout" and is given by a straightforward recursive construction.

2.2 Concepts and Examples

In an object spreadsheet, data is arranged in *cells*. Cells are arranged in columns, and both cells and columns follow a hierarchy of ownership: each column has exactly one *parent column*, and each cell has exactly one *parent cell*, obeying the commutativity rule that a cell's parent always resides in the parent column of the column containing

the cell. The only exception is a specially designated *root column*, which contains exactly one cell (the *root cell*); neither the root column nor the root cell has a parent.

Columns are of two kinds: *value columns* and *object columns*. The cells in value columns (*value cells*) contain primitive values, or references to object cells; cells in object columns (*object cells*) do not store data but instead represent distinct *object identities*, visualized as bullets. To illustrate this, Fig. 2-1 shows a small data-set of students enrolled in various classes next to a list of houses at the school. "Class" is an object column, whereas "name", "student", and "house" are value columns. The root column is at the far left.[1] (We will follow the convention that object column names are capitalized and value column names are written in lowercase.) Child cells of object cells are conceptually owned by the object; child value cells represent fields of the object, while child object cells represent nested objects. Value cells are not allowed to have children.

In the example of Fig. 2-1, "Class" has two child columns, "name" and "student". "Class" and "house" are both directly descended from the root column. Fig. 2-1 presents the columns of this spreadsheet as a tree, expressing the ownership relations. To indicate the column hierarchy visually, we extend the header row so that headers of object columns stretch across those of child columns. (Our tool offers this as a display option.)

Because a cell may have children in more than one column, we group them according to the column to which they belong. The set of all cells in a given column with a given parent object cell is called a *family* and represents the entire value of a field of the parent object, or its entire set of child objects of a certain type. Fig. 2-2 shows the tree of cells for the sheet of Fig. 2-1, with families indicated by forked edges.

Notice that, in the default view, object cells are stretched vertically to span over all rows containing child cells. An entire object and its fields thus occupy a contiguous rectangular region on the spreadsheet area. Value cells normally do not stretch, but when fields are constrained to contain a single value per object,[2] as in the case of the "name" field, it is convenient to match their height with that of the object containing them. The result is a nested table layout.

Conceptually, families are sets. Value cells belonging to the same family all hold distinct values (although a column can have repeating values as long as they occur in different families). The value of a cell never changes during its lifetime; while our UI allows the value in a cell to be edited, semantically such an edit removes the cell and adds a different one to the same family. The items in a family have no intrinsic order, but can be sorted in typical ways in the view (e.g., lexicographic order for strings, chronological order for dates).[3,4]

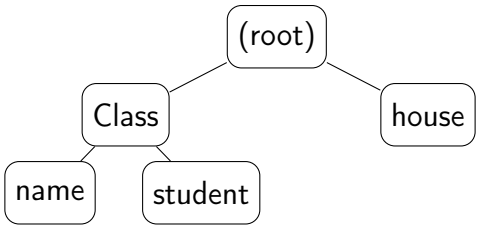
The most basic use case for a spreadsheet is data entry. The owner of a spread-

[1] Consistency would demand a bullet for the root cell, but we hide it because in practice it is mostly just distracting.

[2] We envision this constraint being specified as part of the schema, but our prototype does not yet support this and simply stretches every field that currently contains a single value.

[3] Sorting has not yet been implemented.

[4] To simplify the example in Fig. 2-1, we assume that no two students have the same name; otherwise, a different key has to be used, or a student object column may be introduced.



(root)				
Class				
	name	student		house
•	Charms	Parvati		Gryffindor
•	Potions	Draco		Slytherin
		Lee		Ravenclaw
•	Divination	Lavender		Hufflepuff
		Vincent		
•	Herbology	Neville		

Figure 2-1: A simple spreadsheet with nested elements and its column hierarchy.

sheet, or any user otherwise granted full write access to it, may add cells to any family. Creation of object cells brings to life new families, which are initially empty and can subsequently be extended with new items.

In addition to manual entries, some columns may be populated by computed values that are calculated from other entered (or computed) values. Chapter 3 describes these computations and their semantics.

2.3 Formal Specification

Formally, an object spreadsheet consists of a *schema* Σ defining the structure of the data stored; a *data instance* M containing data conforming to the schema; and a *program* Π containing the formulas for computed fields and the stored procedures. In this section, we describe the structure of the schema and data instance without distinguishing between state and computed data. We introduce that distinction in Section 3.3.2 when we formalize the structure of the program.

A central issue in the formalization of the data model is what object references should consist of, given that Object Spreadsheets allows objects to be computed by formulas. If we use uninterpreted unique identifiers for all objects and allocate arbitrary fresh identifiers for computed objects each time a sheet is evaluated, then "the same" computed object may receive a different identifier when the sheet is reevaluated in response to an unrelated change. As a result, in order to produce a meaningful "diff" of two versions of a spreadsheet or achieve the effect of storing a reference to a computed object in state data,[5] we would need to take extra steps to identify com-

[5]This might seem on its face to be an unlikely thing to do, but developers may choose to define a

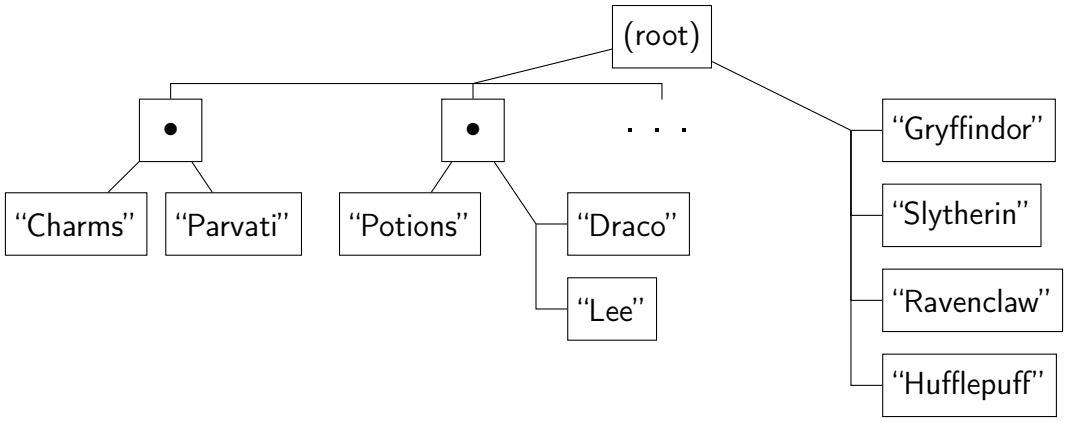


Figure 2-2: Cell hierarchy of the spreadsheet of Fig. 2-1 (only some of the cells are shown). Forked edges indicate cell families.

puted objects based on their role in the sheet. Instead, we design object references to encapsulate exactly the necessary information to identify what we consider to be "the same" object on any version of a spreadsheet, so these tasks do not require any extra effort.[6]

Specifically, each object of a given type T (other than the root object) has an immutable *key* that uniquely identifies it among its parent object's children of type T. Objects in two instances of the spreadsheet that have the same type, the same parent and equal keys are considered to be "the same" object (though possibly in different states). Our design for computed objects (Section 3.1) is such that they inherently have user-defined keys, although one could imagine a system in which keys would be automatically generated in some other stable way. In contrast, the keys for state objects are normally persistent unique identifiers known as *tokens*, which are randomly generated when objects are created and are not user-visible. Our data model also supports state objects with user-defined keys, but our implementation currently does not offer this option because developers found it too hard to understand; we may revisit the issue at some point.

2.3.1 Data Schema

Our tool provides a set of primitive types $\mathbb{P}$, including "text", "number", "boolean", etc. Internally, it also uses a type $\mathcal{T}$ corresponding to an infinite, uninterpreted set of *tokens*, as mentioned above. A function $\mathcal{V}$ is initially defined on each of these types, giving the set of values of that type. If T is a type, we use $x : T$ to mean that x is of

view that provides a more convenient representation of underlying data and then build other parts of an application on top of the view, which would include storing references to the computed objects in the view.

[6]This design increases the learning curve in this section, but the semantics in Section 3.3.2 would be completely unmanageable without introducing it at some point, and introducing it here appears to leave the least mess.

type T, i.e., $x \in \mathcal{V}(T)$. For convenience in some parts of the formalization, we assume that each value is tagged with its type, so that $\mathcal{V}(T)$ and $\mathcal{V}(T')$ are disjoint for every two distinct types T and T'. We also assume a total order on each type, given by a function $\mathsf{systemOrder}$ that takes a finite set of values of the same type and returns a sequence with the values in ascending order.[7]

Definition 1. A *schema* Σ consists of:

- A finite set $\mathbb{C}$ of *columns*, partitioned into a set $\mathbb{OC}$ of object columns and a set $\mathbb{VC}$ of value columns. (Formally, the elements of $\mathbb{C}$ are uninterpreted identifiers that index the columns of the schema. Such identifiers may be reused across schemas, which may be useful when describing a schema editing operation that leaves all but a few columns unaffected. For convenience, we refer to them simply as columns when the meaning is clear.)

- A *root column* $\mathsf{R} \in \mathbb{OC}$ and a *parent function* $\mathsf{p} : \mathbb{C} \setminus \{\mathsf{R}\} \to \mathbb{OC}$ that arranges the columns in a tree rooted at R. We write $D \rightsquigarrow C$ to say that D has a child C (that is, $\mathsf{p}[C] = D$), and also denote $\mathbb{C}^+ = \mathbb{C} \setminus \{\mathsf{R}\}$, $\mathbb{OC}^+ = \mathbb{OC} \setminus \{\mathsf{R}\}$.

- A function $\mathsf{name} : \mathbb{C}^+ \to$ string giving a name to each non-root column.

- A type assignment $\mathsf{type} : \mathbb{C} \to \mathbb{T}$ (where $\mathbb{T} = \mathbb{P} \cup \mathbb{OC}$) such that $\mathsf{type}[C] = C$ for every $C \in \mathbb{OC}$.

- A set $\mathbb{OC}_K \subseteq \mathbb{OC}^+$ of *keyed object columns* (i.e., object columns with user-defined keys[8]) and a function $\mathsf{kc} : \mathbb{OC}_K \to \mathbb{VC}$ mapping each such column C to a *key column* $\mathsf{kc}[C]$ that is one of its children. Let $\mathbb{KC} = \mathsf{kc}[\mathbb{OC}_K]$ be the set of key columns. We define the *key type* $\mathsf{ktype}[C]$ of a column $C \in \mathbb{OC}^+$ to be $\mathsf{type}[\mathsf{kc}[C]]$ if $C \in \mathbb{OC}_K$ or $\mathcal{T}$ otherwise. □

As suggested by the definition of $\mathbb{T}$, each object column C serves as a type, where $\mathcal{V}(C)$ is the set of *references* of the proper format to refer to objects in the column. A reference to an object is composed of the keys of the object and of its ancestors. That is, we define $\mathcal{V}$ on $\mathbb{OC}$ as follows:

- $\mathcal{V}(\mathsf{R}) = \{\mathsf{r}\}$ where r is a unique *root object*.

- $\mathcal{V}(C) = \mathcal{V}(\mathsf{p}[C]) \times \mathcal{V}(\mathsf{ktype}[C])$ for $C \in \mathbb{OC}^+$. We define the projections $\mathsf{p}(x, k) = x$ and $\mathsf{k}(x, k) = k$ where $(x, k) \in \mathcal{V}(C)$.

[7] While the formula language (if extended with ordered lists) could offer any number of explicit sorting functions, one may as well make as reasonable a choice of $\mathsf{systemOrder}$ as possible, which we suggest would consist of the traditional orders on primitive types and lexicographic order on object references, but we do not assume this.

[8] Recall from the beginning of Section 2.3 that computed object columns will always be keyed, while state object columns normally will not be, once we introduce the distinction between state and computed data in Section 3.3.2.

$\mathcal{V}$ can be thought of as an inductive type family, so that if there are cycles in the relation on object columns induced by p and ktype, then $\mathcal{V}$ is simply empty for the affected columns. Of course, such a cycle is probably a mistake and should be reported to the developer, but we want to provide as much well-defined functionality as possible in the (hopefully temporary) presence of errors of any kind.

2.3.2 Data Instance

Definition 2. A *family identifier* for a given schema Σ is a tuple of the form $\langle C, d\rangle$ where $C \in \mathbb{C}^+$ and $d : \mathsf{p}(C)$. □

Definition 3. An *instance* M of a schema Σ consists of:

- A finite set $\mathbb{F}$ of identifiers of families that are defined in the instance.
- A family content function fc that maps each $\langle C, d\rangle \in \mathbb{F}$ to a finite subset of $\mathcal{V}(C)$.

The set of objects that exist in the instance is defined by:

$$\mathsf{objs} = \{\mathsf{r}\} \cup \bigcup_{\substack{C \in \mathbb{OC}^+ \\ \langle C,d\rangle \in \mathbb{F}}} \mathsf{fc}[\langle C, d\rangle]$$

An instance must satisfy the following properties:

- All families belong to existing objects: for every $\langle C, d\rangle \in \mathbb{F}$, $d \in \mathsf{objs}$.
- Families in object columns can only contain objects with the correct parent: if $C \in \mathbb{OC}^+$ and $x \in \mathsf{fc}[\langle C, d\rangle]$, then $\mathsf{p}(x) = d$.
- Keys appear in the key columns: for every $C \in \mathbb{OC}_K$ and every $d : C$ such that $d \in \mathsf{objs}$, $\langle \mathsf{kc}[C], d\rangle \in \mathbb{F}$ and $\mathsf{fc}[\langle \mathsf{kc}[C], d\rangle] = \{\mathsf{k}(d)\}$. (This property, in combination with the first, fully determines the content of key columns.) □

The correspondence between this definition and the informal one of Section 2.2 is that the sheet contains the cell of the root object r, plus for each $\langle C, d\rangle \in \mathbb{F}$ and $x \in \mathsf{fc}[\langle C, d\rangle]$, a cell of value x in column C whose parent is the cell representing d (as a member of $\mathsf{fc}[\langle \mathsf{p}[C], \mathsf{p}(d)\rangle]$, except for $d = \mathsf{r}$, which does not belong to a family). The value of an object cell is a reference to the object itself, which is why $\mathsf{type}[C] = C$ for $C \in \mathbb{OC}$; this convention allows the same definition of down navigation (Section 3.1) to produce the behavior we want for both value columns and object columns. An object with a given reference $o : C$ exists in the instance if and only if $o = \mathsf{r}$ or o appears in the family $\langle C, \mathsf{p}(o)\rangle$ of its putative parent $\mathsf{p}(o)$.

While an object column C contains exactly the objects of type C that exist, a value column C' such that $\mathsf{type}[C'] = C$ may contain arbitrary references of type C to objects that do or do not exist; we call references to objects that do not exist *broken*. Our system does not provide a way to create or compute a broken reference

directly, but an existing reference can become broken when the target object ceases to exist. The reference becomes usable again if an object with the same parent and key comes into existence; this can happen if the key was user-defined, but normally cannot happen if it was a randomly generated token, except perhaps via an "undo" command. Attempting to follow a broken reference raises an error, but broken references can still be counted and tested for equality. While the alternatives of treating broken references as if their targets were empty or even immediately deleting references when they become broken might lead to the desired result in some applications, the behavior we choose seems least likely to result in surprises.

Definition 4. An instance is said to be *complete* in a column $C \in \mathbb{C}^+$ if $\langle C, d\rangle \in \mathbb{F}$ for every $d : \mathsf{p}[C]$ such that $d \in \mathsf{objs}$. □

Normally, we expect an instance to be complete in all columns. However, in Chapter 3, we will discuss the automatic computation of families from formulas, which results in an instance that is incomplete anywhere this computation fails.

Chapter 3

Formulas and Computation

3.1 Concepts

Formulas are assigned to columns rather than to individual cells as they are in traditional spreadsheets. If a column is assigned a formula, it will not admit data entry by typing into the cells—instead, cells in that column are populated by evaluating a formula that computes values based on other data in the spreadsheet. We refer to these columns as *computed columns*, to distinguish them from columns containing mutable state, which we call *state columns*. The formula of a computed column is evaluated once for each object cell in the parent column (known as the *context* object cell for the purpose of the evaluation), generating a family of result cells with that object cell as parent.

The most important aspect of formulas in spreadsheets is access to data in other cells of the spreadsheet. Excel and other traditional spreadsheets use a row-column coordinate notation that is either absolute or relative. Since our data model is hierarchical, data access must follow this hierarchy—a process we refer to as *navigation*.

With a particular cell as the starting point—by default, all navigations start at the context object cell—we distinguish two types of navigation:

- Up navigation—following the parent relationship to go to one of the cell's ancestors.
- Down navigation—retrieving all children of the cell in a particular child column, which comprise a child family of the starting cell. In general, a down navigation results in a set of cells. Since developers like to use short column names that describe the meaning of a column with respect to its immediate parent (e.g. "name", "age"), we allow each single navigation to go down only one level so that its meaning is clear. (To go down more than one level, the developer can chain navigations.)

For simplicity of design, all formula expressions evaluate to sets, following the style of Alloy [25] and DAPLEX [36]. Navigation is done using the *dot notation*: *cell*. *targetName*, where *targetName* is the name of the ancestor or child column, and is naturally lifted to sets by taking the union of the results of the navigation on each

```
expr    ::=  var-name                                  // local variable
         |   (expr .)? column-name ( [expr] )?         // column navigation
         |   literal
         |   { expr* }                                 // union
         |   op expr                                   // unary operator
         |   expr op expr                              // binary operator
         |   function-name ( expr* )                   // built-in function invocation
         |   { var-name : expr | expr }                // filter comprehension
         |   sum[var-name : expr](expr)                // sum comprehension
literal ::=  $                                         // root cell literal
         |   number                                    // singleton number literal
         |   "string"                                  // singleton string literal
         |   true | false                              // singleton Boolean literal
```

Figure 3-1: Formula syntax.

element. *targetName* is resolved to a column when the formula is entered, and the formula is automatically updated if the column is later renamed by the developer, much as a cell reference in a traditional spreadsheet formula updates if the target cell moves.

To resolve the column name in a navigation, we must know the type of the starting cell, therefore formulas are type-checked when they are entered. Type-checking ensures that every subexpression evaluates to a homogeneous set, meaning that all elements are all of the same type (which may be a primitive type or a reference type). As in Alloy, scalar values are represented by singleton sets.

Formula expressions are drawn from a language whose syntax is described in Fig. 3-1. It includes set equality (=) and inclusion (`in`) operators, which can also be used for scalar equality and scalar membership in a set; a set comprehension notation {*var* `:` *set* | *pred*} that filters an existing set using a predicate; and various numeric, boolean, date, and set-related operators and functions (+, −, <, >, `&&`, `count`, etc.).

Our tool currently supports a `sum` construct that binds a variable, for reasons discussed in Section 3.2 example 3. The notation is inspired by the mathematical $\sum$ notation:

$$\sum_{p \in \mathsf{Part}} p.\mathsf{price} \quad \sim \quad \texttt{sum[p : Part](p.price)}$$

When a formula is evaluated, an implicit variable `this` is bound to the context object (the parent of the family being computed). This allows for more compact formulas when accessing fields of the same object and of containing objects. For example, if we were to add a column named `size` as a child column of `Class`, and assign to it the formula `count(students)`, the formula would evaluate to the number of students in each class. In contrast, the formula `count($.Class.students)` would evaluate to the overall number of students (using the special literal `$` that refers to the root cell).

A formula may be assigned:

- To a value column, in which case each value in the set returned by the formula

generates a cell with that value, or

- To an object column, in which case each value generates an object (with a unique identity), and the value itself is stored as a single child in a designated *key column*. One must keep in mind that the formula context is the parent object column, not the object column containing the formula (which is only populated *after* the formula has been evaluated). This feature is analogous to the "COMPOUND OF" function in DAPLEX [36].

Since a key column is associated with the formula of its parent object column, it cannot have a formula of its own.

3.2 Computing With Sets: Examples

We present a series of examples to shed light on how our data model and formula language accommodate the types of set computations that commonly occur in web applications.

1. Given a set `members` of members of a research group (as references to `Person` objects), compute the set of their offices (assumed to be a field `office` of each person). We handle this by allowing a navigation expression on a set of object references, `members.office`, which takes the union of the result for each object in the set.

2. Compute the set of cars currently available in a car sharing service, assuming that each car has an `available` field. This could be achieved by defining a computed field on each car that contains a reference to the car itself if it is available or otherwise the empty set:

   ```
   Car.selfIfAvailable ≙ if(available, Car, {})
   ```

   and then navigating from the set of all cars (`$Car`) to this field:

   ```
   availableCars ≙ $Car.selfIfAvailable.
   ```

   But this type of filtering (like "`WHERE`" in SQL) is so common that we believe it is worth providing the special syntax `{c : $Car | c.available}`. In the future, we plan to consider other syntaxes that may be easier for new developers to understand, perhaps drawing inspiration from Microsoft's LINQ [5], which is a better fit for our set-based language than SQL.

3. Given a set `members` of members of a research group, compute their average age (assumed to be a field `age` of each person, possibly itself computed from the date of birth). Currently, the navigation expression `members.age` drops duplicate ages and returns a set, so we cannot accurately compute an average based on `members.age`. We would like to allow `members.age` to return

a multiset, which could simply be passed to an aggregation function, but it's unclear what should happen when a formula returning a multiset is assigned to an object column to generate child objects. For now, we provide a special aggregation syntax that binds a variable, so the average age formula would look like:

```
averageAge ≙ sum[m : members](m.age) / count(members)
```

4. Suppose we have a set of users, each with a `location` field giving their current location and a `favorites` field containing a set of references to their favorite restaurants (`Restaurant` objects). When a user visits the web application, we wish to show the distance from their current location to each of their favorite restaurants (assuming that `Restaurant` also has a `location` field). The distance depends on both the user and the restaurant, so it can't be defined simply as a computed field on one or the other. One way we can represent the distances is to change the representation of each user's favorites from a set of `Restaurant` references to a set of nested `Favorite` objects, each of which contains a `Restaurant` reference. Indeed, Object Spreadsheets provides a command to perform this conversion. When a user adds or removes a favorite, we create or delete a `Favorite` object rather than just adding or removing a `Restaurant` reference. We can then define a `distance` field on `Favorite` using a formula like:

   ```
   Favorite.distance ≙ dist(User.location, restaurant.location)
   ```

   (imagining for the purpose of this example that `dist` is built-in).

5. Given a set of users and their current locations, when a user logs into the car sharing service (example 2 above), show the distance to each available car. We potentially need a distance for each pair of a user and a car. As in the previous example, we could represent the distance as a `distance` field on a `UserCarInfo` object type that is a child of `User`. But unlike in the previous example, users do not manually add cars to a list that they want to consider, so we cannot create the `UserCarInfo` objects that way. Instead, we can make the `UserCarInfo` object type computed, with a formula that retrieves the set of all available cars. Then, for each user, we will automatically get one `UserCarInfo` object per available car. (In fact, the previous example could alternatively be handled with a computed object type as well.)

   Computed object types can express the same kinds of complex queries that SQL subqueries can, but they maintain the local nature of computation of the spreadsheet paradigm. We suggest that requiring complex queries to be broken down using multiple computed object types may be helpful for their maintainability.

6. Given a set `members` of members of a university department, with a field `city` giving the city in which each lives, compute the set who live in each city for

further analysis. In SQL, this would be done with "`GROUP BY`". In Object Spreadsheets, one would define a top-level computed object type `CityGroup` with formula `members.city`, which would generate one `CityGroup` for each city with at least one resident. One would define a computed field for the set of people in the city as follows:

```
CityGroup.residents ≙ {m : members | m.city = city}
```

Further computed fields can then be added to `CityGroup`. This boilerplate is tolerable for now; we eventually plan to add a special "group by" feature to automate it.

3.3 Computation Semantics

3.3.1 Type Checking

Each time the schema or a formula is modified, the system type-checks all formulas. Specifying the type of a computed column is optional; if unspecified, the type will be inferred during each type-checking pass. This inference is currently implemented as a simple traversal of the formula, recursing on references to other columns (coincidentally, analogous to inference of function return types in TypeScript), so it fails if there is a cycle of (static) dependencies among columns of unspecified type; in that case, the developer can break the cycle by specifying one of the types. For as long as a column's formula cannot be type-checked or fails to match the type specified by the developer (if any), evaluation of families in the column is postponed.

The formal type-checking rules are listed in Fig. 3-2. The schema Σ is fixed and implicit throughout the rules. Formulas are checked in a type environment $\Gamma = \{\texttt{this} \mapsto C\}$ for the appropriate parent column C. The type environment for subformulas may be extended by variable bindings, as in the set comprehension and sum constructs. The helper function lu gives the name lookup rules for up and down navigation. In surface syntax, a subexpression consisting of an identifier id can either refer to a local variable in Γ or represent a navigation $\texttt{this}.id$ (if $\mathsf{lu}(\Gamma[\texttt{this}], id) \neq \{\}$); ambiguity between the two is an error. The `this` variable cannot be referenced explicitly in surface syntax; as a special case, to refer to the context object itself, we require the developer to write its column name C (which represents the no-op up-navigation $\texttt{this}.C$) to remind readers of the formula of its type. For simplicity, in the type-checking rules and the remainder of the semantics, we work with the abstract syntax in which identifiers that represent navigations on `this` have already been expanded into those navigations. Note that an ambiguous navigation in the abstract syntax (for which lu returns more than one interpretation) is ill-typed according to the NAV rule.

When the system parses a formula, it replaces identifiers that refer to columns with the internal column IDs; when the formula is later displayed, these IDs are replaced by the current column names. In this way, columns can be renamed without breaking existing formulas. Parsing of a navigation $e.id$ fails if id does not refer to a

$$\mathsf{lu}(C, id) = \{\langle D, \downarrow\rangle \mid C \rightsquigarrow D,\ \mathsf{name}[D] = id\} \cup \{\langle B, \uparrow\rangle \mid B \rightsquigarrow^* C,\ \mathsf{name}[B] = id\}$$

$$\frac{\Gamma \vdash e : C \quad \mathsf{lu}(C, id) = \{\langle D, dir\rangle\}}{\Gamma \vdash e\,\texttt{.}\,id : \mathsf{type}[D]}\ (\mathsf{NAV})$$

$$\frac{v \in \Gamma}{\Gamma \vdash v : \Gamma[v]}\ (\mathsf{VAR})$$

$$\frac{\forall i \in \{1..n\}.\ \Gamma \vdash e_i : T}{\Gamma \vdash \{e_1\texttt{,}\, e_2\texttt{,}\, \cdots\texttt{,}\, e_n\} : T}\ (\mathsf{SET})$$

$$\frac{\Gamma \vdash e_1 : T, e_2 : T}{\Gamma \vdash e_1 = e_2 : \text{bool},\ e_1\ \texttt{in}\ e_2 : \text{bool}}\ (=/\mathsf{IN})$$

$$\frac{\Gamma \vdash e_1 : \text{numeric}, e_2 : \text{numeric}}{\Gamma \vdash e_1 + e_2 : \text{numeric}}\ (+)$$

$$\frac{\Gamma \vdash e : T \quad \Gamma, v : T \vdash c : \text{bool}}{\Gamma \vdash \{\ v : e \mid c\ \} : T}\ (\mathsf{SET\text{-}COMP})$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{count(}e\texttt{)} : \text{numeric}}\ (\mathsf{COUNT})$$

$$\frac{\Gamma \vdash e : T \quad \Gamma, v : T \vdash a : \text{numeric}}{\Gamma \vdash \texttt{sum[}v : e\texttt{]\,(}a\texttt{)} : \text{numeric}}\ (\mathsf{SUM})$$

$$\frac{}{\Gamma \vdash \texttt{true} : \text{bool},\ \texttt{false} : \text{bool},\ num : \text{numeric},\ \texttt{\$} : \mathsf{R}}\ (\mathsf{LIT})$$

Figure 3-2: Formula type-checking rules. R denotes the root column.

unique column or the type of e cannot be determined in order to resolve id. The same idea applies to the determination of whether a subexpression consisting of an identifier represents a local variable or a navigation on `this`. Once parsed, a formula continues to work even if identifiers in its (current) display representation are ambiguous with respect to the current schema, but the developer will not be able to save an edited version of the formula in which the ambiguity is still present. We do not model these aspects of the system behavior in the formal semantics.

Since all formula values are sets, a judgment $\Gamma \vdash e : T$ means that expression e evaluates (in environment Γ) to a *set* of elements of type T. Each built-in operator and function has its own typing rule. Many have fixed parameter and return types, but not all: for example, the = operator takes two parameters of the same (arbitrary) type and its return type is bool. The rules for =, `in`, +, and `count` are shown as examples. Operators that require scalar arguments, such as +, check at runtime that their arguments are singleton sets.

3.3.2 Sheet Evaluation: Preliminaries

In this section, we begin the formal semantics for sheet evaluation. While the hierarchical structure of the sheet and some of the language constructs are specific to Object Spreadsheets, a large chunk of the semantics would be the same for any system of units that are reactively defined in terms of one another. No one seems to have bothered to publish a formal semantics for a spreadsheet before except for the Deductive Spreadsheet [12]; we discuss this comparison in Section 3.3.3.1.

First we need a few definitions, building on the components of a schema defined in Section 2.3.1.

Definition 5. A *program* Π for a schema Σ consists of a set $\mathbb{CC} \subseteq (\mathbb{VC} \setminus \mathbb{KC}) \cup \mathbb{OC}_K$ of *computed columns* and a function $\Phi : \mathbb{CC} \to \text{expr}$ giving the formulas of these columns. (A program may also include stored procedures, which are described informally in Chapter 4.) The set of *state columns* is defined by $\mathbb{SC} = \mathbb{C}^+ \setminus (\mathbb{CC} \cup \mathbb{KC})$. Families in state columns and computed columns are called *state families* and *computed families*, respectively. To avoid cases where a computed object disappears and descendant state data becomes orphaned, programs must have the property that no state column is a descendant of a computed column. □

Definition 6. A program Π is *well-typed* for a schema Σ if the formula for each computed column C returns the type of its *formula output column* $\mathsf{foc}[C]$, assuming that `this` has the type of the parent column $\mathsf{p}[C]$. The formula output column is C itself if C is a value column, or the key column $\mathsf{kc}[C]$ if C is a keyed object column. Formally, for each $C \in \mathbb{CC}$, we must have $\{\texttt{this} \mapsto \mathsf{p}[C]\} \vdash \Phi[C] : \mathsf{type}[\mathsf{foc}[C]]$ according to the rules in Section 3.3.1. (Our system actually allows the type of $\mathsf{foc}[C]$ to be left unspecified in the schema and sets it to the type inferred for the formula as described in Section 3.3.1, but we do not model that process here.) □

To focus on the important issues, we assume in this discussion that the entire program is well-typed, though our implementation simply evaluates all well-typed computed columns.

Definition 7. A *state instance* M of a schema Σ with a program Π is an instance that is complete (Definition 4) in all state columns and contains no computed families. □

Evaluation begins from a state instance M of a schema Σ with a well-typed program Π. We consider the schema and program fixed and leave them implicit; we do not discuss properties that involve more than one variant of a schema or program here, though some would be straightforward to prove.

3.3.3 The Sheet Monad and Derivations

The simplest approach to sheet evaluation semantics would be to write an evaluation function that takes a family identifier and recurses when the formula reads another family. It has the advantage that determinism is obvious, but two problems:

1. The definition would be logically unsound due to potentially infinite recursion among families.

2. It does not give us the family dependencies explicitly, which we need in order to formalize evaluation algorithms that cache results at the family level, detect cyclic dependencies (Section 3.3.5), and reevaluate the sheet incrementally based on dependencies (Section 3.3.6).

Fortunately, it's almost as easy to write an evaluation function that returns a computation in a *sheet monad* in which reading another family is a built-in operation. This approach solves both problems and still leaves determinism (fairly) obvious.

We define an inductive type T_{sheet} of monadic computations of return type T, as follows:

$$T_{\mathsf{sheet}} := \mathsf{return}\ T \mid \mathsf{err} \mid \mathsf{bindRead}(\langle C, d\rangle, f)$$

The intuitive meaning of the computation $\mathsf{return}\ v$ is to return v immediately, while err raises an error. $\mathsf{bindRead}(\langle C, d\rangle, f)$ reads the value of the family $\langle C, d\rangle$, which must be a finite set of elements of type $\mathsf{type}[C]$, and passes this set to f, which must return a follow-up computation of type T_{sheet}. (That is, the type of f must be $\mathsf{FiniteSet}(\mathsf{type}[C]) \to T_{\mathsf{sheet}}$.)

To execute a computation, if it has the form $\mathsf{bindRead}(\langle C, d\rangle, f)$, we simply pass the value of the family $\langle C, d\rangle$ to f and repeat the process on the follow-up computation until we ultimately reach a computation of the form $\mathsf{return}\ v$ or err. By defining the type T_{sheet} as inductive in a metalanguage that places the appropriate well-foundedness constraints on recursive definitions, we enforce that this process terminates after finitely many steps. We say computations are *locally terminating*: local in the sense that we are considering only the computation itself and not the process used to determine the value of each of the families it reads. If we needed to allow locally nonterminating computations, we could define T_{sheet} as coinductive.

We can define both a read operation (which simply returns the value read) and the monad bind operation in terms of $\mathsf{bindRead}$ as follows:

$$\mathsf{read}(\langle C, d\rangle) = \mathsf{bindRead}(\langle C, d\rangle, (\lambda v.\mathsf{return}\ v))$$

$$\mathsf{bind}(c, g) = \begin{cases} g(v), & c = \mathsf{return}\ v \\ \mathsf{err}, & c = \mathsf{err} \\ \mathsf{bindRead}(\langle C, d\rangle, (\lambda v.\mathsf{bind}(f(v), g))), & c = \mathsf{bindRead}(\langle C, d\rangle, f) \end{cases}$$

In the next section, we will define a function $\mathsf{famcomp}_M$ that gives the computation of type $\mathsf{FiniteSet}(\mathsf{type}[C])_{\mathsf{sheet}}$ used to actually evaluate a family $\langle C, d\rangle$ to a set of elements of $\mathsf{type}[C]$; it makes the necessary case distinctions among state, key, and computed families and evaluates formulas in the appropriate way for computed families. But first we describe the rest of the sheet monad framework, which is independent of the actual definition of $\mathsf{famcomp}_M$.

We'd like to view the local execution trace of a computation $c \in T_{\mathsf{sheet}}$ as a sequence of family reads, with the intervening pure computation being implicit in the initial

construction of c and the functions f passed to $\mathsf{bindRead}$. To build such a trace, we need to know the value of each of the families read. But we still cannot define a function to determine the value of a family, because of the unrestricted recursion. Instead, we define an inductive type family $M \vdash \langle C, d\rangle \Downarrow v$, members of which are *derivations* that the family $\langle C, d\rangle$ *converges* to the value v, and an analogous inductive type family $M \vdash c \Downarrow v$ for computations. (We use notation as if $M \vdash \langle C, d\rangle \Downarrow v$ and $M \vdash c \Downarrow v$ were inductive predicates, but they must actually be inductive type families because we will write algorithms that inspect the structure of derivations.) A derivation of $M \vdash c \Downarrow v$ embeds derivations that the families read by c converge to particular values. In turn, a derivation of $M \vdash \langle C, d\rangle \Downarrow v$ wraps a derivation of $M \vdash \mathsf{famcomp}_M(\langle C, d\rangle) \Downarrow v$. The full convergence rules:

$$\frac{M \vdash \mathsf{famcomp}_M(\langle C, d\rangle) \Downarrow v}{M \vdash \langle C, d\rangle \Downarrow v}\,(\mathsf{Family}) \qquad \frac{M \vdash c \to^* \mathsf{return}\ v}{M \vdash c \Downarrow v}\,(\mathsf{Return})$$

$$\frac{\bigwedge_{i=1}^{n}\big(M \vdash c_{i-1} \to c_i\big)}{M \vdash c_0 \to^* c_n}\,(\mathsf{Multistep}) \qquad \frac{M \vdash \langle C, d\rangle \Downarrow v}{M \vdash \mathsf{bindRead}(\langle C, d\rangle, f) \to f(v)}\,(\mathsf{Read})$$

On the flip side, we have *coinductive* type families $M \vdash c \Uparrow$ and $M \vdash \langle C, d\rangle \Uparrow$ of derivations that a computation or a family *diverges*. Divergence is the opposite of convergence and includes both errors and nontermination due to an infinite regress of dependencies, cyclic or not.[1] As mentioned above, every computation must reach $\mathsf{return}\ v$ or err after reading finitely many families (again, not considering the evaluation of each of those families), so we do not need a divergence rule for a single computation taking an infinite sequence of Read steps. But the local termination property of a computation is moot if it reads a family that diverges. The divergence rules:

$$\frac{M \vdash \mathsf{famcomp}_M(\langle C, d\rangle) \Uparrow}{M \vdash \langle C, d\rangle \Uparrow}\,(\mathsf{FamilyDiv}) \qquad \frac{M \vdash c \to^* \mathsf{err}}{M \vdash c \Uparrow}\,(\mathsf{Err})$$

$$\frac{M \vdash c \to^* \mathsf{bindRead}(\langle C, d\rangle, f) \quad M \vdash \langle C, d\rangle \Uparrow}{M \vdash c \Uparrow}\,(\mathsf{ReadDiv})$$

For convenience, we sometimes speak of the *outcome* of a computation or a family as being either $\Downarrow v$ or $\Uparrow$.

Complexity in the derivation rules costs us throughout the metatheory of derivations, while most of this metatheory is independent of the definition of $\mathsf{famcomp}_M$, so we keep the derivation rules to a bare minimum and move as much of the complexity to the definition of $\mathsf{famcomp}_M$ as possible. We go ahead and give the definitions that motivate the structure of the rules, although we do not use them until Section 3.3.6:

Definition 8. The *local part* of a derivation H consists of the steps from the conclusion up to, but not including, any Family or $\mathsf{FamilyDiv}$ steps (other than the conclusion itself if H is a derivation about a family). The *dependency list* of H, denoted $D(H)$, is the list of unique family identifiers that appear in the hypotheses of Read and $\mathsf{ReadDiv}$

[1] We will see in Section 3.3.5 that an acyclic infinite regress of dependencies cannot occur in the current system, but can occur with an extension that we are considering.

steps in the local part, in the order of their first appearance in the sequence of Read steps followed by the divergence hypothesis of ReadDiv (if applicable). □

An example convergence derivation for a tiny spreadsheet is shown in Fig. 3-3. Here width and height are state columns and area is a computed column with the formula width × height. The state instance M is given by $M.\mathsf{fc}[\langle \mathsf{width}, \mathsf{r}\rangle] = \{4\}$ and $M.\mathsf{fc}[\langle \mathsf{height}, \mathsf{r}\rangle] = \{5\}$. For this example, the definition of $\mathsf{famcomp}_M$ given in the next section reduces to the following:

$$\mathsf{famcomp}_M(\langle C, \mathsf{r}\rangle) = \mathsf{return}\ M.\mathsf{fc}[\langle C, \mathsf{r}\rangle] \quad \text{for } C \in \{\mathsf{width}, \mathsf{height}\}$$

$$\mathsf{famcomp}_M(\langle \mathsf{area}, \mathsf{r}\rangle) = \mathsf{bindRead}(\langle \mathsf{width}, \mathsf{r}\rangle, \lambda x.\mathsf{bindRead}(\langle \mathsf{height}, \mathsf{r}\rangle, \lambda y.\mathsf{return}\ \{s(x) \cdot s(y)\}))$$

Here s returns the single element of a singleton set; to simplify the example, we are ignoring the possibility of an error in s.

The derivation rules have the following important property:

Proposition 1. With respect to a given state instance M, every computation c and family $\langle C, d\rangle$ has a unique derivation of either convergence or divergence, i.e., computations are deterministic.

Proof. The basic idea is that starting from a computation c, the unique derivation will take as many steps as possible using the Read rule (as mentioned above, an infinite sequence of steps is impossible) to arrive at a computation c', then apply Return, Err, or ReadDiv depending on whether c' is of the form return v, err, or $\mathsf{bindRead}(\langle C, d\rangle, f)$ where no derivation of $M \vdash \langle C, d\rangle \Downarrow v$ exists. In the ReadDiv case, coinduction is used to fill in the derivation of the hypothesis $M \vdash \langle C, d\rangle \Uparrow$ of ReadDiv. However, it's in general undecidable whether a derivation of $M \vdash \langle C, d\rangle \Downarrow v$ exists, so the existence proof is nonconstructive. A convergence or divergence derivation for a family $\langle C, d\rangle$ is simply a wrapper around a convergence or divergence derivation for $\mathsf{famcomp}_M(\langle C, d\rangle)$. The existence and uniqueness proofs can be formalized using a mixture of induction and coinduction; they contain no ideas specific to Object Spreadsheets, so we do not go into further detail. □

When we speak informally of the dependencies of a family $\langle C, d\rangle$ with respect to a state instance M, we mean the families in the dependency list of the unique derivation of $M \vdash \langle C, d\rangle \Downarrow v$ or $M \vdash \langle C, d\rangle \Uparrow$.

3.3.3.1 Related Work

The general idea to use a monad to break direct recursion in a definition while still writing it in a similar style is first clearly articulated by Danielsson [18], though some previous work uses the technique. In the "partiality monad" framework of [18], a monadic computation of return type T (including all recursive calls) reduces to a *partial value* of the coinductive type

$$T_\perp := \mathsf{return}\ T \mid \mathsf{later}\ T_\perp.$$

width	height	area
4	5	20

(a) The spreadsheet.

$$\dfrac{\dfrac{\dfrac{\dfrac{\textit{empty list}}{M \vdash c_w \to^* \mathsf{return}\ \{4\}}\,(\mathsf{Multistep})}{M \vdash c_w \Downarrow \{4\}}\,(\mathsf{Return})}{M \vdash \langle \mathsf{width}, \mathsf{r}\rangle \Downarrow \{4\}}\,(\mathsf{Family})}{M \vdash c_a \to c'_a}\,(\mathsf{Read}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\textit{empty list}}{M \vdash c_h \to^* \mathsf{return}\ \{5\}}\,(\mathsf{Multistep})}{M \vdash c_h \Downarrow \{5\}}\,(\mathsf{Return})}{M \vdash \langle \mathsf{height}, \mathsf{r}\rangle \Downarrow \{5\}}\,(\mathsf{Family})}{M \vdash c'_a \to \mathsf{return}\ \{20\}}\,(\mathsf{Read})$$

$$\dfrac{\dfrac{M \vdash c_a \to^* \mathsf{return}\ \{20\}}{M \vdash c_a \Downarrow \{20\}}\,(\mathsf{Return})}{M \vdash \langle \mathsf{area}, \mathsf{r}\rangle \Downarrow \{20\}}\,(\mathsf{Family})\ (\mathsf{Multistep})$$

$$c_w = \mathsf{famcomp}_M(\langle \mathsf{width}, \mathsf{r}\rangle) = \mathsf{return}\ \{4\}$$

$$c_h = \mathsf{famcomp}_M(\langle \mathsf{height}, \mathsf{r}\rangle) = \mathsf{return}\ \{5\}$$

$$f_a = \lambda x.\mathsf{bindRead}(\langle \mathsf{height}, \mathsf{r}\rangle, \lambda y.\mathsf{return}\ \{s(x) \cdot s(y)\})$$

$$c'_a = f_a(\{4\}) = \mathsf{bindRead}(\langle \mathsf{height}, \mathsf{r}\rangle, \lambda y.\mathsf{return}\ \{4 \cdot s(y)\})$$

$$c_a = \mathsf{famcomp}_M(\langle \mathsf{area}, \mathsf{r}\rangle) = \mathsf{bindRead}(\langle \mathsf{width}, \mathsf{r}\rangle, f_a)$$

(b) The convergence derivation H of $\langle \mathsf{area}, \mathsf{r}\rangle$, with auxiliary definitions as shown to reduce clutter in the proof tree. The local part is in the dotted rectangle. The dependency list $D(H)$ is $(\langle \mathsf{width}, \mathsf{r}\rangle, \langle \mathsf{height}, \mathsf{r}\rangle)$.

Figure 3-3: An example convergence derivation for a computed family in a tiny spreadsheet.

A partial value consists of zero or more later steps followed by a return value, or an infinite stream of later steps if the computation does not terminate. We could evaluate families using this framework by defining $\mathsf{read}(\langle C, d\rangle) = \mathsf{later}\ \mathsf{famcomp}_M(\langle C, d\rangle)$; this would solve the recursion, but it would inline the evaluation of a family's transitive dependencies into a single partial value and not give us the dependencies explicitly. The fundamental difference in the sheet monad is that we keep $\mathsf{bindRead}(\langle C, d\rangle, f)$ as a constructor of T_{sheet} and write derivation rules that expose the dependency structure. We also add an err constructor, but it could be folded into return using a Maybe type as in [18] if we wished.

The only previous formal semantics we could find for a spreadsheet system is for the Deductive Spreadsheet by Cervesato [12]. It is based on the concept of an *environment* η, an assignment of values to the cells of the sheet in which cells may take the *uncalculated value* $\perp$. A function $\ddot{E}$ is defined that takes an old environment η and produces a new environment η' in which each cell's formula is evaluated based on the values in η; this evaluation uses special definitions of the language constructs that propagate $\perp$. Then $\ddot{E}$ is iterated starting from the empty environment $\eta_0 = \lambda c.\perp$ to produce environments $\ddot{E}^i(\eta_0)$. Since sheets are finite, cell references in formulas are static, and cyclic dependencies are disallowed, the iteration must reach an environment $\ddot{E}^n(\eta_0)$ in which no cells are uncalculated ([12] section 3.2.1). If these restrictions were not in place, one could still define an (in general uncomputable) final environment

$$\hat{\eta}(c) = \begin{cases} x, & \exists i : \ddot{E}^i(\eta_0)(c) = x \neq \perp \\ \perp, & \text{otherwise.} \end{cases}$$

The Deductive Spreadsheet also supports logic programs, which are automatically broken into strata, each of which acts as a single unit in the sheet evaluation process ([12] section 5.5.2).

Our formulation is simpler in that it uses inductive types to abstract away the iteration and a monad to abstract away the propagation of $\perp$ in the definitions of the language constructs. One might feel that the difference is unimportant because the behavior is intuitively clear based on either formulation. Indeed, the behavior may be intuitively clear without a formal semantics at all. The objective of a formal semantics should be to make completely formal proofs easiest to write (if one were to pursue them, e.g., using a proof assistant), and our formulation is certainly preferable in that regard.

Cervesato does discuss a semantics based on an inductive predicate for the logic programs embedded in a deductive spreadsheet ([12] section 5.3.5), and it is possible to convert a spreadsheet to a logic program, but we cannot consider Cervesato to have applied this semantics to the entire spreadsheet.

3.3.4 Family Evaluation

With the sheet monad in place, we proceed to give the definition of $\mathsf{famcomp}_M$ in Fig. 3-4. The definition uses Haskell-style "do" notation, in which "**do** $x \leftarrow c_1$; c_2" denotes $\mathsf{bind}(c_1, \lambda x.c_2)$, and so forth, where bind is as defined in the previous section.

This "glue" code makes all of the case distinctions that are external to formula evaluation: state families are looked up from the state instance, key families are generated automatically, and computed families are evaluated using their formulas, with keys converted to computed object references in computed object columns (a conversion that is reversed in those objects' key families). $\mathsf{checkObj}(d)$ is a helper function that checks that the object d exists in the evaluated sheet. The definitions of checkObj and the generation of key families in $\mathsf{famcomp}_M$ match the definitions of $M.\mathsf{objs}$ and the key families in M with respect to state columns, but also work for computed columns.

For state families, $\mathsf{famcomp}_M$ consults M directly, which is why M is passed as an argument; since $M.\mathbb{F}$ contains the correct set of families, a call to checkObj would be redundant and we omit it. In all other cases, $\mathsf{famcomp}_M$ delegates to compute, which may read families via the monad but does not access M directly. (This fact will become important in Section 3.3.6.)[2]

As mentioned in Section 2.3.2, there is no way to compute a broken reference directly, so the only way a call to $\mathsf{famcomp}_M$ on a family of an existing object can lead to a checkObj failure (or the "$C \in \mathbb{SC}$ and $\langle C, d\rangle \notin M.\mathbb{F}$" case) is if a formula attempts to follow a reference that was previously stored in a state column and has since become broken.

To complete the definition of $\mathsf{famcomp}_M$, we define the structurally recursive denotation function $[\![e]\!]\sigma$ for formulas in Fig. 3-5 and Fig. 3-6. σ is an environment that maps local variables to their values, which must be finite sets. The definition is written in terms of a "parallel bind" operation, $\mathsf{pbind}((c_i)_{i=1}^n, f)$, that takes a list of subcomputations and passes the list of their results to f. We extend the "do" notation so that a use of $\leftarrow$ with a list of subcomputations represents a call to pbind. For the purpose of sheet evaluation, we simply give pbind sequential semantics, i.e.,

$$\mathsf{pbind}((c_i)_{i=1}^n, f) = \mathsf{bind}(c_1, \lambda v_1. \cdots \mathsf{bind}(c_n, \lambda v_n. f((v_i)_{i=1}^n)) \cdots),$$

since this is the simplest way to get deterministic divergence derivations. Nevertheless, we define $[\![e]\!]\sigma$ in terms of pbind to emphasize the logical structure of the computation. The p and systemOrder functions used in the definition are described in Section 2.3.1.

The denotation rules for most language constructs should be unsurprising. Note that up-navigation on an object reference raises an error if the target of the reference no longer exists, even though the (former) ancestor can be computed syntactically from the object reference. This is to prevent broken references from violating developer assumptions that if all objects in a column C with a given ancestor o have been deleted (or formulas have been defined not to generate any), then up-navigation on an object reference of type C cannot return o.

The definition of $\mathsf{famcomp}_M$ together with all its helper functions has two important properties:

- All sets that arise in the definition are finite, so systemOrder is applied only

[2]Since the definition of an instance requires that M contain key families for the state columns, in principle we could also consult M directly for these key families. However, it's probably simpler for implementations not to maintain key families as part of a state instance and instead use compute for all key families, as indicated in our definition of $\mathsf{famcomp}_M$.

$$\mathsf{famcomp}_M(\langle C, d\rangle) = \begin{cases} \mathsf{return}\ M.\mathsf{fc}[\langle C, d\rangle], & C \in \mathbb{SC} \text{ and } \langle C, d\rangle \in M.\mathbb{F} \\ \mathsf{err}, & C \in \mathbb{SC} \text{ and } \langle C, d\rangle \notin M.\mathbb{F} \\ \mathsf{compute}(\langle C, d\rangle), & C \in \mathbb{KC} \cup \mathbb{CC} \end{cases}$$

$$\mathsf{compute}(\langle C, d\rangle) = \left\{ \begin{array}{ll} \mathbf{do} & \_ \leftarrow \mathsf{checkObj}(d) \\ & \left\{ \begin{array}{l} \text{if } C \in \mathbb{KC}: \quad \mathsf{return}\ \{\mathsf{k}(d)\} \\ \text{else } (C \in \mathbb{CC}): \\ \left\{ \begin{array}{ll} \mathbf{do} & v \leftarrow [\![\Phi[C]]\!]\{\mathtt{this} \mapsto d\} \\ & \mathsf{return} \begin{cases} \{(d,k) \mid k \in v\}, & C \in \mathbb{OC}_K \\ v, & \text{otherwise} \end{cases} \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

$$\mathsf{checkObj}(d) = \left\{ \begin{array}{l} \text{if } d = \mathsf{r}: \quad \mathsf{return}\ () \\ \text{else } (d : C \in \mathbb{OC}^+): \\ \left\{ \begin{array}{ll} \mathbf{do} & S \leftarrow \mathsf{read}(\langle C, \mathsf{p}(d)\rangle) \\ & \begin{cases} \mathsf{return}\ (), & d \in S \\ \mathsf{err}, & \text{otherwise} \end{cases} \end{array} \right\} \end{array} \right\}$$

Figure 3-4: "Glue" code for family evaluation that makes all of the case distinctions external to formula evaluation.

$$[\![v]\!]\sigma = \mathsf{return}\ \sigma[v] \quad \text{if } v \text{ is a variable}$$

$$\mathsf{single}(c) = \left\{ \begin{array}{ll} \mathbf{do} & S \leftarrow c \\ & \begin{cases} \mathsf{return}\ v, & S = \{v\} \\ \mathsf{err}, & \text{otherwise} \end{cases} \end{array} \right\}$$

$$[\![e\,.\,id]\!]\sigma = \left\{ \begin{array}{ll} \mathbf{do} & v \leftarrow [\![e]\!]\sigma \\ & \mathbf{let}\ (v_i)_{i=1}^n = \mathsf{systemOrder}(v) \\ & (S_i)_{i=1}^n \leftarrow (\mathsf{navigate}(v_i, id))_{i=1}^n \\ & \mathsf{return}\ \bigcup_{i=1}^n S_i \end{array} \right\}$$

$$\mathsf{navigate}(o, id) = \begin{cases} \left\{ \begin{array}{ll} \mathbf{do} & \_ \leftarrow \mathsf{checkObj}(o) \\ & \mathsf{return}\ \mathsf{ancestor}(o, C) \end{array} \right\}, & \text{if } o : D,\ \mathsf{lu}(D, id) = \{\langle C, \uparrow\rangle\} \\ \mathsf{read}(\langle C, o\rangle), & \text{if } o : D,\ \mathsf{lu}(D, id) = \{\langle C, \downarrow\rangle\} \end{cases}$$

$$\mathsf{ancestor}(o, C) = \begin{cases} o, & o : C \\ \mathsf{ancestor}(\mathsf{p}(o), C), & \text{otherwise} \end{cases}$$

Figure 3-5: Monadic denotational semantics for formulas, part 1.

$$[\![\{\, e_1, \ldots, e_n \,\}]\!]\sigma = \left\{ \begin{array}{l} \textbf{do} \quad (S_i)_{i=1}^n \leftarrow ([\![e_i]\!]\sigma)_{i=1}^n \\ \qquad \mathsf{return}\ \bigcup_{i=1}^n S_i \end{array} \right\}$$

$$[\![e_1 = e_2]\!]\sigma = \left\{ \begin{array}{l} \textbf{do} \quad (v_1, v_2) \leftarrow ([\![e_1]\!]\sigma, [\![e_2]\!]\sigma) \\ \qquad \mathsf{return}\ (v_1 = v_2) \end{array} \right\}$$

$$[\![e_1 \texttt{ in } e_2]\!]\sigma = \left\{ \begin{array}{l} \textbf{do} \quad (v_1, v_2) \leftarrow ([\![e_1]\!]\sigma, [\![e_2]\!]\sigma) \\ \qquad \mathsf{return}\ (v_1 \subseteq v_2) \end{array} \right\}$$

$$[\![e_1 + e_2]\!]\sigma = \left\{ \begin{array}{l} \textbf{do} \quad (S_1, S_2) \leftarrow ([\![e_1]\!]\sigma, [\![e_2]\!]\sigma) \\ \qquad (x, y) \leftarrow (\mathsf{single}(\mathsf{return}\ S_1), \mathsf{single}(\mathsf{return}\ S_2)) \\ \qquad \mathsf{return}\ \{x + y\} \end{array} \right\}$$

$$[\![\{x \texttt{:} e \mid c\}]\!]\sigma = \left\{ \begin{array}{l} \textbf{do} \quad S \leftarrow [\![e]\!]\sigma \\ \qquad \textbf{let}\ (v_i)_{i=1}^n = \mathsf{systemOrder}(S) \\ \qquad (b_i)_{i=1}^n \leftarrow (\mathsf{single}([\![c]\!](\sigma\{x \mapsto v_i\})))_{i=1}^n \\ \qquad \mathsf{return}\ \{v_i \mid 1 \le i \le n,\ b_i = \mathit{true}\} \end{array} \right\}$$

$$[\![\texttt{count}(e)]\!]\sigma = \left\{ \begin{array}{l} \textbf{do} \quad S \leftarrow [\![e]\!]\sigma \\ \qquad \mathsf{return}\ |S| \end{array} \right\}$$

$$[\![\texttt{sum[}x\texttt{:}e\texttt{] (}a\texttt{)}\,]\!]\sigma = \left\{ \begin{array}{l} \textbf{do} \quad S \leftarrow [\![e]\!]\sigma \\ \qquad \textbf{let}\ (v_i)_{i=1}^n = \mathsf{systemOrder}(S) \\ \qquad (u_i)_{i=1}^n \leftarrow (\mathsf{single}([\![a]\!](\sigma\{x \mapsto v_i\})))_{i=1}^n \\ \qquad \mathsf{return}\ \sum_{i=1}^n u_i \end{array} \right\}$$

$$[\![\texttt{if}(e_1, e_2, e_3)]\!]\sigma = \left\{ \begin{array}{l} \textbf{do} \quad b \leftarrow \mathsf{single}([\![e_1]\!]\sigma) \\ \qquad \begin{cases} [\![e_2]\!]\sigma, & b = \mathit{true} \\ [\![e_3]\!]\sigma, & b = \mathit{false} \end{cases} \end{array} \right\}$$

$[\![l]\!]\sigma = \mathsf{return}\ \{l\}$ if l is a primitive literal

$[\![\$]\!]\sigma = \mathsf{return}\ \{\mathsf{r}\}$

Figure 3-6: Monadic denotational semantics for formulas, part 2.

to finite sets and pbind only to finite sequences, as required. This is easiest to see by applying a "type checking" point of view to the definition, assuming that $\mathsf{read}(\langle C, d\rangle)$ has type $\mathsf{FiniteSet}(\mathsf{type}[C])_{\mathsf{sheet}}$. Families in M are finite by the definition of an instance, and no construct in the formula language can generate an infinite set from finite sets.

- $\mathsf{famcomp}_M(\langle C, d\rangle)$, like any computation in T_{sheet} defined by well-founded (or as a special case, structural) recursion, must terminate after a finite number of local reads.

Having completed the definition of family evaluation, we can now describe the result of evaluating the entire sheet:

Definition 9. The *computed instance* of M, denoted $\widehat{M}$, consists of the families and values given by family convergence derivations of the form $M \vdash \langle C, d\rangle \Downarrow v$. (By Proposition 1, the value of a given family is unique. It is straightforward to show based on the definition of $\mathsf{famcomp}_M$ that $\widehat{M}$ is a valid instance, using Proposition 2 below for the finiteness.) □

All readers of the data model, including the spreadsheet UI and web application views, use the computed instance. Families $\langle C, d\rangle$ such that $d \in \widehat{M}.\mathsf{objs}$ but $\langle C, d\rangle \notin \widehat{M}.\mathbb{F}$ are those that failed to evaluate. The spreadsheet UI marks them specially, but it's unclear what is the best thing to do in general when a web application view includes such a family.

We decline to expose any information about the cause of an error in divergence judgments: our intent is that implementations have a debugger that shows the full execution trace represented by the derivation, and our implementation does. In general, developers may need the full execution trace to identify where behavior began to differ from their expectations. Implementations may choose to show additional information about errors up front as a convenience, including information that crosses family boundaries, e.g., "cyclic dependency between families $\langle C_1, d_1\rangle$ and $\langle C_2, d_2\rangle$". We do not model this in the semantics, especially because cross-family information would complicate the statement of Proposition 3, but we point out that such error reporting inherits the determinism of the underlying derivations.

Determinism of errors may appear to be a minor point, given that one hopes that errors rarely occur. Its practical significance (in conjunction with the soundness of dependency tracking) is that if the developer makes a change to the sheet while debugging an error in a family $\langle C, d\rangle$ and the execution trace of $\langle C, d\rangle$ changes, the developer can be sure that their change actually affected $\langle C, d\rangle$ according to the semantics and did not merely trigger a different nondeterministic choice in the implementation.

3.3.5 The Evaluation Algorithm and Computability

The archetypal algorithm to evaluate a sheet, shown in Fig. 3-7, is essentially a constructive analogue of Proposition 1. It assumes a fixed state instance M and

populates a cache $\mathcal{H}$ that maps family identifiers to convergence or divergence derivations. The evaluateFamily function can be called to evaluate individual families on demand as they are viewed in the spreadsheet UI or a web application view. If desired, the evaluateAll function of Fig. 3-8 can be used to force evaluation of the entire sheet. To detect dependency cycles, the algorithm maintains an explicit stack S of the calls to evaluateFamily, which is empty between top-level calls. Each entry on S is a pair of a family identifier $\langle C, d\rangle$ and a multi-step derivation of the form $M \vdash \mathsf{famcomp}_M(\langle C, d\rangle) \to^* \mathsf{bindRead}(\langle C', d'\rangle, f)$, where $\langle C', d'\rangle$ is the target of the next call to evaluateFamily. We use the names of the derivation rules from Section 3.3.3 throughout as constructors that take derivations of the hypotheses and return a derivation of the conclusion.

The algorithm is written to generate derivations to make its correctness clear, but it should be clear that we can erase the derivations and keep only the outcome of each family and the results will still be correct. The spreadsheet debugger can always regenerate the local part of the derivation of a given family on demand using the cached outcomes.

The algorithm detects dependency cycles, but it fails to terminate if called on a family with an infinite, acyclic forward chain of dependencies. We envision that a product-quality implementation would evaluate the sheet in the background, allowing the developer to continue working and canceling the evaluation if the state instance changes, or would give up at an implementation-defined limit, possibly giving the developer the option to continue further.

Nontermination cannot actually happen in the data model as currently defined because evaluated sheets are finite:

Proposition 2. There are finitely many objects d that exist (in the sense that checkObj(d) succeeds).

Proof. It suffices to show this property for objects in each column C. The proof is by induction on the depth of C in the column tree. An object d exists according to checkObj if it is the root or the family $\langle C, \mathsf{p}(d)\rangle$ converges and it is a member of that family. Recall that the content of a family is always a finite set.

If $C \in \mathbb{SC}$, then the claim follows immediately from the finiteness of M. Otherwise, by the inductive hypothesis, $\mathsf{p}[C]$ contains finitely many objects, and the convergent families of these objects in column C together contain finitely many objects. □

Corollary 1. During a top-level call to evaluateAll (Fig. 3-8) or to evaluateFamily on an arbitrary family identifier, evaluateFamily is called on finitely many different families.

Proof. evaluateSubtree only calls evaluateFamily directly on families of existing objects, and the only objects that are available for formulas to attempt to read their families are existing objects and the targets of broken references already stored in the state instance. By Proposition 2, there are finitely many families of existing objects. Since state instances are finite, there are finitely many broken references, and attempting to follow one of them can at most cause checkObj to recurse through

$\mathcal{H} \leftarrow \{\}$; $S \leftarrow []$

function evaluateFamily($\langle C, d\rangle$)
  if $\mathcal{H}[\langle C, d\rangle] = \mathsf{null}$ **then**
    $\mathcal{H}[\langle C, d\rangle] \leftarrow \mathsf{evaluateFamilyImpl}(\langle C, d\rangle)$
  end if
  return $\mathcal{H}[\langle C, d\rangle]$
end function

$\mathsf{evaluateFamilyImpl} \leftarrow \mathsf{evaluateFamilyFull}$ // Hook for the incremental algorithm

function evaluateFamilyFull($\langle C, d\rangle$)
  $c \leftarrow \mathsf{famcomp}_M(\langle C, d\rangle)$
  $L \leftarrow []$
  loop
    if c is of the form $\mathsf{return}\ v$ **then**
      return $\mathsf{Family}(\mathsf{Return}(\mathsf{Multistep}(L)))$
    else if $c = \mathsf{err}$ **then**
      return $\mathsf{FamilyDiv}(\mathsf{Err}(\mathsf{Multistep}(L)))$
    else // c is of the form $\mathsf{bindRead}(\langle C', d'\rangle, f)$
      $H' \leftarrow \mathsf{readFamily}(\langle C, d\rangle, \mathsf{Multistep}(L), \langle C', d'\rangle)$
      if H' diverges **then**
        return $\mathsf{FamilyDiv}(\mathsf{ReadDiv}(\mathsf{Multistep}(L), H'))$
      end if
      // Otherwise conclusion of H' is of the form $M \vdash \langle C', d'\rangle \Downarrow v'$
      $c \leftarrow f(v')$
      $L.\mathsf{append}(\mathsf{Read}(H'))$
    end if
  end loop
end function

function readFamily($\langle C, d\rangle, T, \langle C', d'\rangle$)
  $S.\mathsf{push}(\langle C, d\rangle, T)$
  if $\langle C', d'\rangle$ is on S **then**
    $(\langle C_i, d_i\rangle, T_i)_{i=1}^{n} \leftarrow$ stack entries starting from $\langle C_1, d_1\rangle = \langle C', d'\rangle$ to the top
    $H' \leftarrow$ cyclic derivation H_1 defined by
      $H_i = \mathsf{FamilyDiv}(\mathsf{ReadDiv}(T_i, H_{(i+1) \bmod n}))$ for $i = 1, \ldots, n$
  else
    $H' \leftarrow \mathsf{evaluateFamily}(\langle C', d'\rangle)$
  end if
  $S.\mathsf{pop}()$
  return H'
end function

Figure 3-7: The full evaluation algorithm.

```
function evaluateSubtree(d)
    C ← type of d
    for each child column C′ of C do
        H ← evaluateFamily(⟨C′, d⟩)
        if conclusion of H is of the form M ⊢ ⟨C′, d⟩ ⇓ v then
            for d′ ∈ v do
                evaluateSubtree(d′)
            end for
        end if
    end for
end function

function evaluateAll
    evaluateSubtree(r)
end function
```

Figure 3-8: Code to force evaluation of the entire sheet.

finitely many nonexistent ancestors. Similarly, if evaluateFamily is called on an arbitrary family identifier $\langle C, d\rangle$, then either d exists or checkObj recurses through finitely many nonexistent ancestors. □

In fact, it can be shown that the functions currently computable using Object Spreadsheets are precisely the primitive recursive functions, if an argument n is given in the form of the set $\{0, \ldots, n-1\}$. We could add a built-in function to generate such sets, but it has not been a priority. Instead, we are considering adding a feature to the system that would enable general recursion, namely, the ability for any formula to construct "virtual" objects with arbitrary values for their state fields and then read their computed fields. Virtual objects would be analogous to the nested sheets that represent user-defined functions in Forms/3 [11] and the spreadsheet extension of Peyton Jones et al. [34]. In the remainder of the discussion, we imagine such an extension has been made, so the definition of an instance no longer requires finiteness (though in a real implementation, state instances would have to be finite) and Proposition 2 no longer holds. Then, a simple example of nontermination would occur with a virtual object type that represents a user-defined function definition $f(x) \mathrel{\widehat{=}} f(x+1)$.

3.3.6 Dependency Tracking and Incremental Reevaluation

We have said nothing yet about the fact that the outcome (return value or divergence) of a family is determined by the outcomes of its dependencies. This *locality* property of spreadsheet-like systems is valuable for developer understanding and program analysis and makes it sound for implementations to reevaluate the sheet incrementally after the values of some state families are changed. First we formalize the property.

Proposition 3. Let M_0 and M be state instances for the same schema and program. Suppose $C \notin \mathbb{SC}$ and H_0 is a derivation of $M_0 \vdash \langle C, d\rangle\ o$, where o is an *outcome* of the form $\Downarrow v$ or $\Uparrow$. Let $(\langle C_i, d_i\rangle)_{i=1}^n$ be the dependency list of H_0 (Definition 8), and let o_i be the outcome of $\langle C_i, d_i\rangle$ in M_0. Then a derivation H with respect to M with the same local part as H_0 (except for the replacement of M_0 with M) can be constructed from derivations of $M \vdash \langle C_i, d_i\rangle\ o_i$ for $i = 1, \ldots, n$.

If $o = \Uparrow$, then this construction is coinductive and *guarded*, meaning that the assumed subderivations are used only unmodified in the resulting derivation under at least one new step. The significance of this stipulation is that it is safe to apply the proposition circularly to all of the ReadDiv derivations in a dependency cycle for M_0 to yield an identical cycle for M, provided that all dependencies not part of the cycle have the same outcomes with respect to both state instances.

Proof. Since $C \notin \mathbb{SC}$, we have $\mathsf{famcomp}_{M_0}(\langle C, d\rangle) = \mathsf{compute}(\langle C, d\rangle)$, which does not depend on M_0. Thus, replacing M_0 with M in the local part of H_0 and incorporating the derivations of $M \vdash \langle C_i, d_i\rangle\ o_i$ gives a valid derivation H for M. □

We can write a naive "incremental" algorithm that uses Proposition 3 directly and rescans the whole sheet but avoids the actual reevaluation of families whose dependencies did not change. The code is in Fig. 3-9 as a delta with respect to Fig. 3-7. Here $\mathcal{H}_0$ may be the cache generated by either the incremental algorithm or the full algorithm on a previous state instance. Both algorithms maintain the invariant that if $\mathcal{H}$ contains a derivation of family convergence or divergence, it also contains derivations for the dependencies, except while a dependency cycle is being unwound. The incremental algorithm assumes that $\mathcal{H}_0$ has the same property.

Note that by the time evaluateFamilyIncremental($\langle C, d\rangle$) calls readFamily on a family $\langle C', d'\rangle$, it is known that all previous dependencies have the same outcomes as in H_0, and therefore if we were to reexecute $\mathsf{famcomp}_M(\langle C, d\rangle)$, it would still read $\langle C', d'\rangle$. (This idea is made precise in the construction of L.) This means that the incremental algorithm only evaluates families that the full algorithm would also evaluate, and in particular, it must terminate if the full algorithm does.

The incremental algorithm still works if we erase the derivations and store only the outcome and dependency list of each family. As the next step toward a realistic implementation, we can maintain a persistent cache $\bar{\mathcal{H}}$ with reverse dependency lists. When state families are changed, we mark their transitive dependents as "dirty" and add any of these dependents that are currently being viewed to a work list. At this point, the whole of $\bar{\mathcal{H}}$ represents $\mathcal{H}_0$ in the naive incremental algorithm, while the subset of clean entries (whose full derivations are still valid) represents $\mathcal{H}$. Accordingly, evaluateFamily should call evaluateFamilyImpl if and only if the family does not have a clean entry. Because we are not keeping the old data for families that have already been reevaluated, the read of the old outcome of $\langle C', d'\rangle$ from $\mathcal{H}_0$ must be replaced by a test of a flag on the entry for $\langle C', d'\rangle$ indicating whether its outcome changed in the current pass. To complete the reevaluation, we just call evaluateFamily on each family on the work list and purge any remaining dirty entries, because the single-pass change flags are insufficient for future passes to know whether the outcomes of the dependencies of such entries have changed since the entries were originally generated.

```
evaluateFamilyImpl ← evaluateFamilyIncremental

function evaluateFamilyIncremental(⟨C, d⟩)
    if C ∈ 𝕊ℂ or ℋ₀[⟨C, d⟩] = null then
        return evaluateFamilyFull(⟨C, d⟩)
    end if
    H₀ ← ℋ₀[⟨C, d⟩]
    newDeps ← []
    for ⟨C′, d′⟩ in D(H₀) do
        // H′₀ must exist and match the subderivation of H₀.
        H′₀ ← ℋ₀[⟨C′, d′⟩]
        L ← Read steps of H₀ up to the first read of ⟨C′, d′⟩,
              with dependency derivations replaced with newDeps
        H′ ← readFamily(⟨C, d⟩, Multistep(L), ⟨C′, d′⟩)
        if H′ diverges then
            return FamilyDiv(ReadDiv(Multistep(L), H′))
        else if conclusion of H′ ≠ conclusion of H′₀ then
            return evaluateFamilyFull(⟨C, d⟩)
        end if
        newDeps.append(H′)
    end for
    return H₀ with dependency derivations replaced with newDeps
end function
```

Figure 3-9: The naive incremental evaluation algorithm.

Alternatively, if each entry includes the outcomes of its dependencies from which it was generated, then the change flags are not necessary, and it becomes safe to keep dirty entries in the cache for families that are no longer needed by the current view, in case the families become needed again in the future and the entries are found to be valid again at that time. This is the approach commonly used by build tools such as Pluto [22].[3]

Note that a dirty entry for a family $\langle C, d\rangle$ may be marked clean again without reexecuting $\mathsf{famcomp}_M(\langle C, d\rangle)$ if none of the dependencies ultimately changed. In the worst case, after a change to a single state family, the system might spend time traversing many families to mark them dirty and clean again without their outcomes changing. In a few hours, we were unable to find any promising general approach to mitigate this problem without giving up the property of only evaluating families that the full algorithm would. Of course, one trick that may reduce the traversal overhead without giving up termination guarantees is to speculatively start evaluating the immediate dependents of the changed state family to see if they are unchanged, and after a timeout, mark them dirty and proceed with the traversal.

3.3.7 Determinism and Catching of Cyclic Dependency Errors

By Proposition 1, the derivation (or outcome if we erase the derivation) returned by evaluateFamily for a given family $\langle C, d\rangle$ is the same no matter what sequence of calls is made to evaluateFamily. This is not true of some previous systems that appear to be based on a cycle-detecting evaluation algorithm like evaluateFamily but allow formulas to catch cyclic dependency errors. For example, we observed the following phenomenon in LibreOffice Calc version 5.0.6.1:

1. Define:

   $$\text{A1} = \text{"X"} \,\&\, \mathsf{IFERROR}(\text{B1}, \text{"\_"}), \quad \text{B1} = \text{"Y"} \,\&\, \mathsf{IFERROR}(\text{A1}, \text{"\_"})$$

   The formulas evaluate to "XY_" and "Y_", respectively.

2. Cut and paste A1 to A2. The formulas still evaluate to "XY_" and "Y_".

3. Undo twice and then redo twice. This should leave the sheet in the same state, but now A2 reads "X_" and B1 reads "YX_".

One could at least guarantee determinism by fixing the order in which cells are evaluated, but the property no longer holds that the outcome of a computed cell is the result of its formula on the outcomes of its dependencies. It's not obvious how to write a semantics that is easy to reason about in this (hopefully rare) case. Until then, for some applications, nondeterminism or violations of the locality property might be considered a lesser evil than unconditional application failure. At least there is no

[3]Yet another option is to keep the entire local part of the derivation for each family. This makes it possible to pick up execution of $\mathsf{famcomp}_M(\langle C, d\rangle)$ from the first dependency that changed, saving any work before that point, but this may not be a useful point in the design space compared to deeper analysis of subformula dependencies.

ParentView		Child		Meeting			
	parent		student	enrollment	teacher	sel. slot	avail. slots
	Person		Person	Enrollment	Person	Slot	Slot
•	Molly	•	Fred	• Fred @ Potions	Snape	Snape @ 1pm	
				• Fred @ Divination	Trelawney		Trelawney @ 4pm
							Trelawney @ 5pm
							Trelawney @ 6pm
		•	George	• George @ Charms	Flitwick		Flitwick @ 10am
							Flitwick @ 11am

Hello, Molly.

Meetings for Fred

selected

Snape @ 1pm

[×]

pick one:

[○] Trelawney @ 4pm

[○] Trelawney @ 5pm

[○] Trelawney @ 6pm

Meetings for George

pick one:

[○] Flitwick @ 10am

[○] Flitwick @ 11am

Figure 3-10: A view model and one instance for scheduling parent-teacher meetings, with an example rendering.

problem in allowing errors unrelated to cyclic dependencies to be caught: they can be viewed as a new kind of convergent value for a cell.

3.4 Application Views

Application views follow MVC guidelines. The design of HTML views is hierarchical by nature, so it seems desirable to have a hierarchical model backing it. We define a *view model* to be a designated sub-tree of the data model, by picking an object column $C \in \mathbb{OC}$ and including all its descendant columns and all the cells in these columns. The view model is crafted through formulas to contain exactly those data items that are to be displayed. If a view needs some parameters, such as the currently logged-in user, selected class, etc., then these parameters are placed in state fields of a common *view instance* object $v : C$. This allows several instances of the same view to exist simultaneously.

The view instance is then mapped onto an HTML template using standard templating techniques. Notice that at this point the template does not have to contain any logic such as conditional statements; such logic can be pushed to the formulas populating the view. This makes the binding straightforward, following the nested structure of the view model.

An example from the parent-teacher conference application described in Section 5.2 is shown in Fig. 3-10. Under the object column "ParentView", "parent" is a state value column that is filled with a reference to the user requesting the view.

Formulas then pull out the relevant data from the other columns in the data model. Notice that "ParentView"⇝"Child" and "Child"⇝"Meeting", matching the containment structure of the rendered HTML.

To understand how the control aspect works, notice the buttons [×] and [○] in the figure; clicking a button fires a transaction that mutates the data in order to schedule or cancel a meeting. Transactions are explained in the next chapter; the important thing to notice is that the button is contained in a UI element, which is in turn associated with a cell in the spreadsheet, simplifying the task of associating the click with the relevant data item(s) that need to be updated.

Chapter 4

Transactions

An object spreadsheet can contain *transaction procedures*, which are stored procedures that can be called by unprivileged users to mutate the spreadsheet state, similar to events in Sunny [30]. Each execution of such a procedure is a *transaction*. Transactions are atomic with respect to readers and other transactions and are automatically rolled back if they fail.

Transaction procedures are written in a simple procedural language that currently does not include all the abstraction capabilities of general-purpose programming languages; we may consider offering greater abstraction to advanced developers in the future. Like the language of DAPLEX [36], our language requires separate statements to create an object and initialize each of its fields, and it offers statements to update sets incrementally by additions and removals. The grammar is shown in Fig. 4-1. Each procedure has a signature defined by a type assignment Γ to named parameters. The body of the procedure is a sequence of statements.

Much of the semantics is self-explanatory. For all of the mutation statements, if the first expression returns multiple objects, each is mutated in the manner described. Families of (value) cells are manipulated as sets of values, via `:=`, `add`, and `remove`. State objects are created with the `new` statement, which returns a new object each time it executes; `delete` deletes all objects returned by the expression, including all data they own.

The `check` statement fails the transaction if the condition is false. An idiom is to define a cell whose formula is the conjunction of all application-specific data validity conditions and `check` it at the end of each transaction. A transaction also fails if any formula in a statement fails to evaluate. We leave to future work the issue of giving unprivileged users as much information about errors as possible without leaking confidential data.

Like formulas, procedures are type-checked when they are first defined (in order to resolve column references) and again after each change to the schema; a procedure that is ill-typed with respect to the current schema cannot be executed. The `let` statement sets a local variable and adds an entry to the type environment Γ for subsequent statements according to the type of the expression assigned, as determined by the rules of Fig. 3-2. The type of `new` e is the same as that of the corresponding sub-expression e. Since conditional assignment to a local variable is a common programming idiom,

```
procedure  ::=  ( Γ ) → block
block      ::=  statement*
statement  ::=  let var-name = expr                          // set local variable
            |   expr . column-name := expr                   // replace content of value family
            |   to set expr . column-name add expr           // add element(s) to value family
            |   from set expr . column-name remove expr      // remove element(s) from value family
            |   (let var-name =)? new expr . column-name     // create object
            |   delete expr                                  // delete object
            |   if expr { block } (else { block })?
            |   foreach (var-name : expr) { block }
            |   check expr                                   // validation/assertion
```

Figure 4-1: Procedural language syntax.

we make local variable assignments inside an `if` statement visible after the statement, provided that the variable has the same type at the end of both branches; if the types differ, the variable cannot be read after the `if` statement. Assignments inside a `foreach` loop are not visible outside the loop.

Several example transaction procedures are included in the description of the Hack-q application in Section 5.3. Clearly, writing a transaction procedure presents a greater challenge to an end-user developer than writing formulas. For creation, update, and deletion transactions on a single object type, the tool could offer a command to generate a procedure, which the developer could then customize. We envision letting the developer set up one or more example calls with particular arguments and optionally a hard-coded starting state (if the production state changes too rapidly to make a stable example case), and then showing the mutations that would be made and local variables that would be bound by each statement as it is written, an approach known as example-centric programming [20]. To diagnose problems with past transactions, the system could store their execution traces and allow the developer to browse and search these traces as well.

Chapter 5

Experiments and Evaluation

5.1 Prototype Implementation

We have built a prototype of the Object Spreadsheets execution engine and developer interface on top of the Meteor web framework [29]; all our applications thus inherit the reactivity of Meteor. The developer UI is rendered via a Handsontable [24] widget with cell merging managed by our code, and supports editing the schema (that is, the overall structure) and its contents. Formulas and values are type-checked to ensure conformance to the schema. Transaction procedures are executed by the engine, but they cannot yet be edited in the developer interface (so they must be provided in a file).

5.2 Overview of Example Applications

To assess the applicability of our model, we collected scenarios in which our colleagues faced a need for a collaborative data-driven web application for a specific task. We noted a few of the most interesting features of each application and considered how best to implement them in an object spreadsheet. We then built the essential parts of these applications, and hand-coded UIs for them with basic client-side Meteor templates (eventually, UI building will be integrated in the developer interface).

The applications are:

- PTC—the parent-teacher conference application mentioned in the the introduction. Teachers, students, and parents are stored as Person objects. A reference field links students to their parents. Teachers own Slot objects that represent potential meeting times. Classes are stored using another top-level object column, and each class owns Section objects, which in turn have references to teachers teaching those sections, as well as own Enrollment objects that link to enrolled students. Parents can only schedule one meeting per Enrollment of each of their children, in a slot of the correct teacher (i.e., the teacher of the section in which the student is enrolled); and slots cannot be double booked.

- Dear Beta, a site for students working on a system architecture assignment to share advice on correcting particular test failure modes. Students can vote on questions and answers as on Stack Overflow. The questions are organized in a tree structure matching the structure of the exercises given in class.

- Hack-q, a system for participants in a hackathon to request help from mentors in particular areas of expertise. This case study is discussed in more detail below.

- Got Milk, a management application for a group of people who share a pool of fresh milk for coffee. Teams of two members take turns buying the milk for the entire group. The application sends e-mail notifications for members when it is their turn to buy the milk, and alerts when milk supply is low.

To give an idea of the size of the applications, Table 5.1 shows the sizes of the Object Spreadsheets that were used to back them. The numbers under "Data" and "Formulas" indicate the number of columns of respective kind. The numbers under "Procedures" indicate the number of lines of procedure code that were written for mutations.

5.3 Example Application: Hack-q

We present the data model, formulas, and transactions constructed for the "Hack-q" example and explain their function in finer detail. In this application, participants of an organized hackathon access a web form where they fill in their name, the programming area in which they require assistance, and their current location. Meanwhile, designated mentors have been classified according to their area of expertise—each mentor has been assigned one or more "skills". The submitted request then shows up in the relevant mentors' queues as a "call". A mentor can then "pick" the call, in which case it disappears from the queues of other mentors. After talking to the participant, the mentor may close the call (discarding it from the queue), or forfeit the call, putting it back so that it reappears in all other queues and can subsequently be picked up again by another mentor.

Fig. 5-1 shows a sample sheet containing some concrete data. Column names, their types, and their hierarchy are shown by the header of the spreadsheet. The

	Data		Formulas	Procedures
	total # columns	# object columns	# computed columns	LOC
PTC	40	12	9	17
Dear Beta	13	7	1	7
Hack-q	13	3	1	9
Got Milk	16	5	1	16

Table 5.1: Sizes of sample applications.

formula for the column "inbox" (under "Staff") computes a mentor's incoming queue. Every mentor is assigned a set of "Call" objects on subjects relevant to the mentor's skills as listed in the "expertise" column. The calls are sorted according to the "time" column. Calls assigned to other mentors, and calls that have been forfeited by that mentor, are subtracted from their queue. The transactions are used to insert and remove elements from the queue, and are quite straightforward.

For comparison, we built as much of Hack-q in QuickBase as we could. We created Skills and Calls tables as in Fig. 5-1, but we stored the expertise information in reverse by adding a QuickBase user-list field, "Experts", to the Skills table to hold the set of mentors with the skill. With this representation (which we found slightly unnatural), we were able to define an Inbox report on the Calls table that tested whether the current user was in the Experts list of the skill record associated with each call. However, if we wanted to enhance the application so that a call could require multiple skills, this would require only a small change in Object Spreadsheets but we are not aware of a way to express such logic in QuickBase; this illustrates the risk that developers take by investing in a tool with limited expressive power. Also, QuickBase does not support application-specific mutation patterns (such as assigning a call to the current user) in the core application builder; instead, the developer has to write a formula to concatenate strings into a URL that will make the desired change via the QuickBase API and then add a link to this URL to the page.

5.4 User Study

To begin to collect feedback about how a tool like ours would be received by the intended end-user developers, we conducted a small user study, recruiting four individuals with experience building data-driven applications from a local user group for one of the existing application builders and from our department. We guided each participant through the process of building a book marketplace application (very similar in complexity to the parent-teacher conference application) using Object Spreadsheets. With three of the participants, we went through the same process in QuickBase. We wanted to know the strengths and weaknesses the participants experienced in each tool once they understood the basics; we did not believe a learnability test with no guidance would be realistic or appropriate at this stage.

The major lessons we learned:

- One participant said editing the entire schema and data set on one screen was much better than going back and forth between schema and data pages for each table in QuickBase.

- Two of the participants liked the ability to nest objects and said they would use it. One of them has built several applications backed by a data warehouse and expressed frustration with the number of different database views that had to be joined explicitly, and observed that appropriate use of nesting in the original views would reduce this burden. This remark was made in the context

Skill	
	name
	text
•	Python
•	Android
•	Firefox
•	Linux

Staff			
	name	expertise	inbox
	text	Skill	Call
•	Remus	Firefox	Myrtle
		Python	
•	Severus	Linux	Angelina
		Python	Neville
•	Dolores	Android	Angelina
		Linux	Myrtle
		Firefox	

Call						
	time	name	location	issue	assign	forfeit
	date	text	text	Skill	Staff	Staff
•	9:53	Angelina	Forbidden Forest	Linux		
•	10:18	Myrtle	Chamber of Secrets	Firefox		
•	11:31	Neville	Hall of Hexes	Python	Severus	

Formulas

```
inbox    {c : $.Call |
            c.issue in expertise
            && (c.assign = {} || c.assign = {Staff})
            && !(Staff in c.forfeit)}
```

Procedures

```
enqueue  (name : text, issue : text, location : text)
         let q = new $.Call
         q.time := now
         q.name := name
         q.location := location
         q.issue := {s : $.Skill | s.name = issue}
         check q.issue != {}
```

```
pick     (call : Call, user : Staff)
         call.assign := user
```

```
forfeit  (call : Call)
         to set call.forfeit add call.assign
         call.assign := {}
```

```
done     (call : Call)
         delete call
```

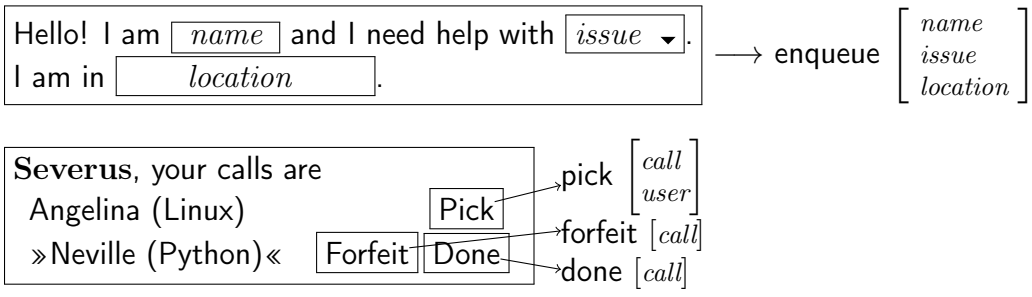


Figure 5-1: Data model, formulas, transaction procedure code, and HTML forms associated with the simple queuing example "Hack-q" in the case study.

of querying an existing database, but one would imagine the same principle would apply to designing an application from scratch.

- All participants struggled to write formulas, especially set comprehensions. The ability of end-user developers to construct formulas is critical for the success of our system, and we believe the situation will improve with the implementation of the formula builder.

In addition, participants pointed out a number of specific UI elements that were confusing or error-prone and made suggestions for improvement, which we are in the process of reviewing.

Chapter 6

Related Work

Spreadsheet-backed application builders. The only spreadsheet-backed application builder designed to allow persistent state in the spreadsheet to be mutated via the application UI is Quilt [10]. It uses an unmodified Google Spreadsheet and does not attempt to overcome the limitations of the traditional spreadsheet model, so at most it supports a single table containing one record per row. The developer creates an HTML page and specifies an element to be repeated to display each record, subelements of which may be bound to fields of the record using column names or may be hidden conditionally based on fields. In addition, controls can be designated to add and delete records.

Spreadsheet design features have been pursued more extensively for development of "mashup" applications that combine, transform, and query data from multiple sources but do not maintain their own persistent state. Gneiss [13, 14] and SpreadMash [26] both retain the two-dimensional grid but allow cells to contain nested data structures retrieved from external sources. They can extract and filter items from these structures, and SpreadMash can even define computed fields on them, but neither tool can generate or mutate such structures on its own.

Other data-driven application builders. Subtext with Two-Way Dataflow [19] is analogous to our work in offering a continuously visible rich data model with computed data and application UI binding support, and it has an intriguing design for batching view updates using a total order on the data model. However, its developer UI is less familiar than a spreadsheet, and it's unclear to us how the UI will scale to development of our target applications.

App2You [27] and AppForge [39] both let the developer build hierarchical forms, constructing the schema automatically, and offer a menu of access control policies. However, neither has demonstrated how end-user developers would build arbitrary logic. AppForge supports only filters of the form "field operator constant", and [27] does not state the form of filters supported by App2You, though it mentions a formula language as a future extension.

Finally, the mainstream application builders QuickBase [1], FileMaker [2], and Knack [4] all support computed fields that are a function of fields of the same object or aggregations of related objects, but none has the ability to repeat a computation

on each related object, as Object Spreadsheets can by introducing a computed object column.

Naked Objects. Our work may recall the Naked Objects approach to application design [33]. The essential principles of this approach are (1) a commitment to encapsulating all logic in the objects it affects and (2) automatic generation of the application UI from the schema and programmatic interfaces. The use of Object Spreadsheets as a data model and development tool for the application logic appears to be orthogonal to both of these principles. (Object Spreadsheets currently does not provide any features to enforce encapsulation, but a developer can still choose to respect it.) Furthermore, the spreadsheet UI on the application state could play the role of the automatically generated application UI, except for the need for read access control. It does not yet provide a way to invoke procedures, but this is planned.

Nested table interfaces to relational data. Related Worksheets [8] is a spreadsheet-like tool that lets a developer construct a schema for a set of related tables and join them into editable nested-table views. The original vision was to provide most of the features of spreadsheets, but formula support was never added. Instead, the authors went on to develop SIEUFERD [7], a tool for exploring existing relational databases. SIEUFERD lets a developer construct nested-table views using menu commands for joins, filtering, and sorting, but does not support modifying the data or the schema. SIEUFERD also supports computed fields, with a formula language that supports navigations both up and down the hierarchy, though many data transformations are only achievable via the menu commands. SIEUFERD assembles its views by generating a set of SQL queries and does not have a semantics at the cell level, and indeed, some changes to the view definition have non-local effects that surprised us. We believe there is a subset of SIEUFERD's functionality that (with the use of some boilerplate) is equivalent to the computational capabilities of Object Spreadsheets, though we have not verified this. Object Spreadsheets differs in its support for schema and data editing, stored procedures, and abstract object references and its demonstration as a backend for web applications.

SheetMusiq [28] is similar in computational capabilities to SIEUFERD, but its UI differs superficially from a nested table layout: it repeats the values in columns at outer levels of the nesting.

Mashroom [23] uses a typed, nested data model and a nested table layout like ours and has a formula language similar in spirit to ours with support for hierarchical navigation (Mashroom does not have object references). However, its computational model is based on a script of transformations starting from the source data, such as "insert a column containing a snapshot of the result of this formula", which can then be replayed on new source data. One could achieve a development cycle similar to ours by modifying formulas in the script and replaying it, with the limitation that dependencies must be acyclic at the column level rather than the family level. Also, Mashroom does not consider mutations to a permanent state, as contrasted with edits recorded in the script.

Structured spreadsheets. Several existing spreadsheet tools are able to maintain varying degrees of structure within a two-dimensional grid. In Microsoft Excel, if a formula is entered in a cell in a range designated as a "table", then the column of the range becomes "calculated" and is automatically filled with the same formula (including rows added to the table later) and Excel warns if the formula is overridden in individual cells.

MDSheet [17] is somewhat more general. It lets the developer define the structure and formulas of a spreadsheet using the ClassSheets [21] modeling format, which supports repeating row and column groups and a dot notation for navigation, and it maintains the structure and formulas as the developer adds and removes instances of the repeating groups. There appears to be no obstacle in principle to supporting programmatic addition and removal of such instances. However, MDSheet supports only one level of repetition along each axis of the grid, which limits its expressive power compared to Object Spreadsheets.

Sumwise [31, 32] lets the developer annotate rows and columns with arbitrary tags (which could represent object types or fields) and then declaratively bind formulas to all cells with certain tags. However, the available documentation is insufficient for us to evaluate its expressive power compared to Object Spreadsheets, and it does not make explicit the points at which the developer foresees addition or removal of instances of repeating row or column groups.

Other spreadsheet extensions. We have reviewed several systems that extend spreadsheets with new capabilities to see if they might be capable of handling nested variable-size sets with per-item formulas definable in context, even if their original motivation differed from ours. Mini-SP [40] meets this test, since it allows sheets to be nested in cells and has a rich programming language that includes the ability to instantiate a nested sheet for each cell in an input array, but the code required is more complex than in Object Spreadsheets. Forms/3 [11] is capable of mapping auxiliary sheets containing per-item formulas across a variable-size input array but does not support nested data. The Analytic Spreadsheet Package [35] and the spreadsheet of Clack and Braine [16] combine the two-dimensional grid with formulas in more powerful programming languages that can manipulate nested data structures, so such structures can be stored in a single cell, but items in these structures are not first-class entities in the system as they are in Object Spreadsheets.

Other tools that bridge databases and spreadsheets. Senbazuru [15] automatically recognizes and extracts relational data from existing spreadsheets and provides a UI for developers to perform certain types of queries, but it does not allow queries to be defined persistently, does not match the expressiveness of Object Spreadsheets, and does not have an approach to handle programmatic mutations.

Sroka et al. [37] give a construction to store relational tables in a traditional spreadsheet and execute SQL queries reactively by translating them to intricate spreadsheet formulas. This may be a convenient environment to work with data, but it does not improve upon SQL in terms of end-user development of queries.

Chapter 7

Conclusions and Future Work

Our work so far has demonstrated the basic idea of a spreadsheet with support for structured data and shown how the logic of data-driven web applications can be expressed in such a spreadsheet. Our experience has been that the spreadsheets are easy to understand and easy to change, making them suitable for rapid prototyping without up-front design and for situations where the requirements tend to change frequently. However, work remains to be done to support the complete web application development process and make it as easy as possible for end-user developers. The major areas of future work we envision are as follows. Of course, we plan to continue seeking feedback from our target end-user developers and adjust plans accordingly.

Casual editing: Interaction techniques and editing operations to help developers work toward the appropriate schema for their data set one step at a time, supporting the same casual work style as unstructured spreadsheets rather than placing a premium on thinking at a high level of abstraction in order to get the schema right in advance. In one potential interaction technique, applied to the space allocation example of Section 1.4.1, the developer would initially lay out all the columns without any nesting and enter one room with a single occupant. Then, they would go to add a second occupant and realize that it should have its own cells in the "name" and "role" columns but not in the other columns, so they would select the "name" and "role" cells of the first occupant and invoke a command, "insert cells below". The tool would automatically group the "name" and "role" columns into a nested object type and add a second nested object to the room, with the effect of inserting cells below the originally selected ones.

Configurable axes: Support for laying out different fields and object types as well as different objects of the same type along either axis of the sheet, rather than always laying out fields and object types in columns and different objects of the same type in rows. This would support basic use cases such as totals at the bottom of a column and grids with one variable on each axis as well as more complex layouts with arbitrary groups of cells that repeat along each axis, as in MDSheet [17].

Formula builder: An interactive formula builder analogous to that of a traditional spreadsheet tool (with integrated language documentation and subexpression results) but addressing the issues unique to our data model. Such issues include (1) visualizing all forms of navigation expressions the system may ultimately support and constructing them by clicking; and (2) previewing the result of a subexpression for each of several different values of the bound variables.

Procedure builder: A procedure builder analogous to the formula builder to develop the procedures that implement web application mutations.

Model and language extensions: Enhancements to the data model and language to make application logic easier to express. These could include additional options for the content of a family (a single value, multiset, or ordered list rather than a set); a "group by" operator; and "virtual" objects that are passed by value and do not have their own permanent locations in the state (though copies of them can be stored inside other objects). Any formula could construct a virtual object by specifying values for its "state" fields and then read its computed fields. Virtual objects could be used to represent invocations of user-defined functions (as in Forms/3 [11] and the spreadsheet extension of Peyton Jones et al. [34]) as well as application view instances (Section 3.4), neither of which should have their own permanent existence in the state.

Updatable views: Support for writing to a "view" consisting of computed data, either directly via the spreadsheet or application UI or programmatically, to trigger writes to underlying data. A write may be implemented by "running the formula in reverse" when doing so has a clear meaning or via a custom procedure or some other programming model. It may be difficult to implement the desired update behavior when writes to a view arrive one field at a time, so we may need an abstraction to batch writes before they are sent to the update procedure, or we may use a more rigidly structured model like Subtext's Two-Way Dataflow [19] if case studies suggest it will work well.

UI builder: A builder for the application UI that includes all the features of the spreadsheet interface, but applied to a WYSIWYG representation of the page instead of the nested table layout. For example, to design a page in which the user enters a query parameter in a field at the top and sees a list of matching objects in a nested table, the developer would enter the query formula directly into the nested table in the UI preview, clicking on the field to refer to the parameter value entered by the user.

Programmatic interfaces to other services: Support for Object Spreadsheets applications to interact with other services by publishing and consuming time-varying data sets or by sending or receiving events or remote procedure calls. Some care is needed to integrate such interactions tastefully into the programming model,

maintaining as much of its predictability as possible. Probably the most ubiquitous example of an interaction is that applications need to send email. The interface to send a message could consist of a procedure call or of insertion of a row into a special table, but the latter design raises the questions of whether the table stores sent messages indefinitely and what happens if one attempts to delete a row for a message already sent.

Bibliography

[1] Business apps, online databases & custom software: Intuit QuickBase. http://quickbase.intuit.com/.

[2] Create custom solutions: FileMaker. http://www.filemaker.com/.

[3] Google Forms - create and analyze surveys, for free. http://www.google.com/forms/about.

[4] Knack - easy online database and business apps. https://www.knackhq.com/.

[5] LINQ (Language-Integrated Query). https://msdn.microsoft.com/en-us/library/bb397926.aspx.

[6] Online form builder with cloud storage database: Wufoo. http://www.wufoo.com/.

[7] Eirik Bakke and David R. Karger. Expressive query construction through direct manipulation of nested relational results. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, New York, NY, USA, 2016. ACM. To appear; preprint available at http://people.csail.mit.edu/ebakke/research/sieuferd_sigmod2016.pdf.

[8] Eirik Bakke, David R. Karger, and Rob Miller. A spreadsheet-based user interface for managing plural relationships in structured data. In *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011*, pages 2541–2550, 2011.

[9] Eirik Bakke, David R. Karger, and Robert C. Miller. Automatic layout of structured hierarchical reports. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2586–2595, 2013.

[10] Edward Benson, Amy X. Zhang, and David R. Karger. Spreadsheet driven web applications. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 97–106, New York, NY, USA, 2014. ACM.

[11] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Funct. Program.*, 11(2):155–206, March 2001.

[12] Iliano Cervesato. *The Deductive Spreadsheet*. Springer Berlin Heidelberg, 2013.

[13] Kerry Shih-Ping Chang and Brad A. Myers. Creating interactive web data applications with spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 87–96, New York, NY, USA, 2014. ACM.

[14] Kerry Shih-Ping Chang and Brad A. Myers. A spreadsheet model for using web service data. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pages 169–176, July 2014.

[15] Zhe Chen, Michael Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. Senbazuru: A prototype spreadsheet database management system. *Proc. VLDB Endow.*, 6(12):1202–1205, August 2013.

[16] Chris Clack and Lee Braine. Object-oriented functional spreadsheets. In *Proc. 10th Glasgow Workshop on Functional Programming*, GlaFP '97, 1997.

[17] Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. Mdsheet: A framework for model-driven spreadsheet engineering. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1395–1398, Piscataway, NJ, USA, 2012. IEEE Press.

[18] Nils Anders Danielsson. Operational semantics using the partiality monad. In *International Conference on Functional Programming 2012, ACM Press*, pages 127–138, 2012.

[19] Jonathan Edwards. Two-way dataflow. In *Future of Programming Workshop 2014*. `https://vimeo.com/106073134`.

[20] Jonathan Edwards. Example centric programming. *SIGPLAN Notices*, 39(12):84–91, December 2004.

[21] Gregor Engels and Martin 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.

[22] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 89–106, New York, NY, USA, 2015. ACM.

[23] Yanbo Han, Guiling Wang, Guang Ji, and Peng Zhang. Situational data integration with data services and nested table. *Serv. Oriented Comput. Appl.*, 7(2):129–150, June 2013.

[24] A minimalist Excel-like data grid editor for HTML & JavaScript. `www.handsontable.com`.

[25] Daniel Jackson. Alloy: A new technology for software modelling. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, page 20, 2002.

[26] Woralak Kongdenfha, Boualem Benatallah, Régis Saint-Paul, and Fabio Casati. Spreadmash: A spreadsheet-based interactive browsing and analysis tool for data services. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering*, CAiSE '08, pages 343–358, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Keith Kowalczykowski, Kian Win Ong, Kevin Keliang Zhao, Alin Deutsch, Yannis Papakonstantinou, and Michalis Petropoulos. Do-it-yourself custom forms-driven workflow applications. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[28] Bin Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 417–428, Washington, DC, USA, 2009. IEEE Computer Society.

[29] An open source platform for building web applications. `www.meteor.com`.

[30] Aleksandar Milicevic, Daniel Jackson, Milos Gligoric, and Darko Marinov. Model-based, event-driven programming paradigm for interactive web applications. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 17–36, 2013.

[31] Darren Miller, Gary Miller, and Luis M. Parrondo. Sumwise: A smarter spreadsheet. In *EuSpRiG*, 2010.

[32] Gary Miller. The spreadsheet paradigm: A basis for powerful and accessible programming. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 33–35, New York, NY, USA, 2015. ACM.

[33] Richard Pawson. *Naked objects*. PhD thesis, Trinity College, 6 2004.

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

[35] Kurt W. Piersol. Object-oriented spreadsheets: The analytic spreadsheet package. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 385–390, New York, NY, USA, 1986. ACM.

[36] David W. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6(1):140–173, March 1981.

[37] Jacek Sroka, Adrian Panasiuk, Krzysztof Stencel, and Jerzy Tyszkiewicz. Translating relational queries into spreadsheets. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2291–2303, August 2015.

[38] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 360–384, 2014.

[39] Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. Wysiwyg development of data driven web applications. *Proc. VLDB Endow.*, 1(1):163–175, August 2008.

[40] A.G. Yoder and D.L. Cohn. Real spreadsheets for real programmers. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, pages 20–30, May 1994.