MIT-CSAIL-TR-2016-001 January 12, 2016

Initial report on Object Spreadsheets

Richard Matthew McCutchen, Shachar Itzhaky, and Daniel Jackson

Initial report on Object Spreadsheets *

Richard Matthew McCutchen, Shachar Itzhaky, and Daniel Jackson

Massachusetts Institute of Technology
Cambridge, MA, USA
{rmccutch,shachari,dnj}@mit.edu

Updated documents for this project will be available at `http://sdg.csail.mit.edu/projects/objsheets/`.

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.

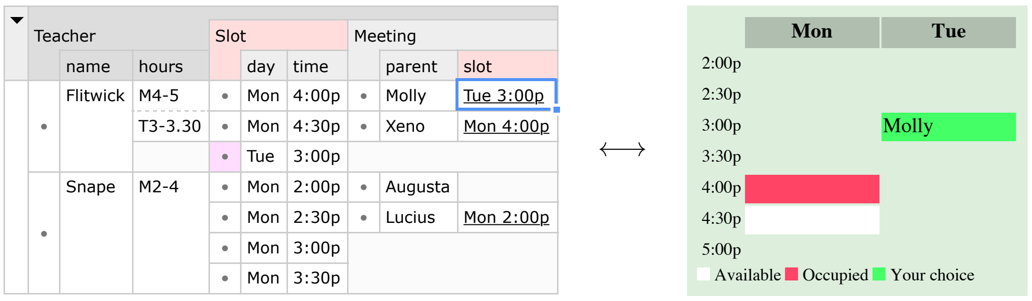
1998 ACM Subject Classification D.1.6 Logic Programming; D.1.7 Visual Programming; H.2.1 Logical Design: Data models; H.2.3 Languages: Database (persistent) programming languages; H.2.3 Languages: Query languages; H.4.1 Office Automation: Spreadsheets

Keywords and phrases end-user development, data-driven web applications, spreadsheets, entity-relationship data

1 Introduction

A wide class of 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).

* 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.



	Teacher		Slot			Meeting		
	name	hours		day	time		parent	slot
•	Flitwick	M4-5	•	Mon	4:00p	•	Molly	Tue 3:00p
		T3-3.30	•	Mon	4:30p	•	Xeno	Mon 4:00p
			•	Tue	3:00p			
•	Snape	M2-4	•	Mon	2:00p	•	Augusta	
			•	Mon	2:30p	•	Lucius	Mon 2:00p
			•	Mon	3:00p			
			•	Mon	3:30p			

⟷

	Mon	Tue
2:00p		
2:30p		
3:00p		Molly
3:30p		
4:00p	🟥	
4:30p	⬜	
5:00p		

⬜ Available 🟥 Occupied 🟩 Your choice

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

Organizations with such a need face a dilemma: to purchase an off-the-shelf solution (which may fail to meet the specific 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 [5] (leaving a considerable burden of manual work).

Ideally, an organization's administrators would build a suitable application by themselves (*end-user development*), but this is rarely done for this type of application. 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 data-driven *web application builders* (such as App2You [24] and Intuit QuickBase [1]) make it easier for people with little or no programming experience to build 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.

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[32, 14], user-defined functions [10, 31], and stream processing [33]. It seems to us that spreadsheets offer similar potential benefits for end-user development of web applications, in particular for the storage and manipulation of persistent data.

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 makes simple data transformations easy (with the ability to select operands visually) and helps the user work through harder ones (with integrated function documentation and immediate feedback on subexpression results).

Obtaining all these advantages in the richer context of the kind of data store and programs required for a web application is not trivial, however (and in the next section we outline particular challenges). In fact, Quilt [9] 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 leverage these advantages while making adaptations to the spreadsheet model that allow it to be applied in a broader context than the traditional one.

In this paper, we propose an enhanced spreadsheet tool, called Object Spreadsheets, that can be used by an end-user developer (henceforth the "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.

In addition to the editable sheet with formulas, Object Spreadsheets supports stored procedures to define the updates that unprivileged 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 potential users of the tool.

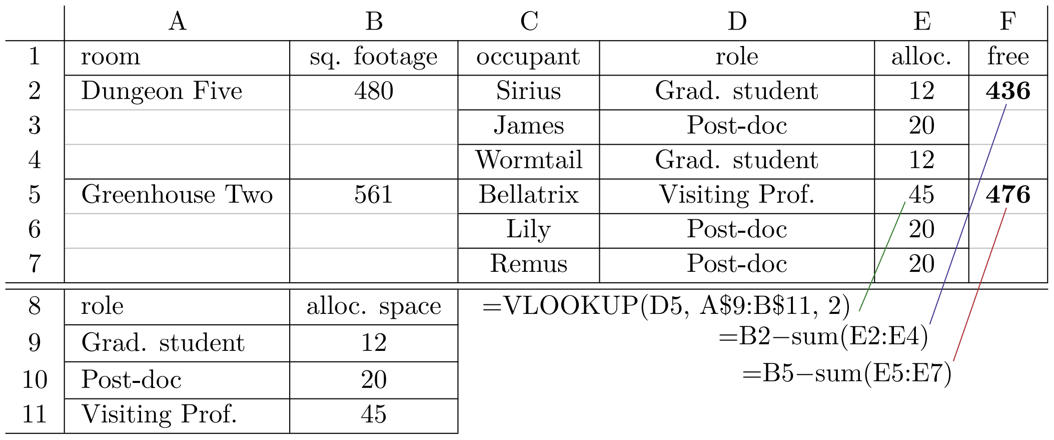
The rest of the paper describes in more detail how the logic of a data-driven web application can be built using Object Spreadsheets. Our contributions include:

- An analysis of the challenges of extending the spreadsheet paradigm to support web application development;
- A data model (Section 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 (Section 3), and a simple procedural language to support restricted updates (Section 4);
- A prototype implementation of the tool (Section 5);
- A suite of example applications that demonstrate common difficulties presented by this application class, and their implementations in our tool (also in Section 5);
- Qualitative feedback from a user study.

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://web.mit.edu/rmccutch/www/relsheet/`. These materials may help the reader quickly get a sense of what Object Spreadsheets does before reading further in the paper.

1.1 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 Section 6.



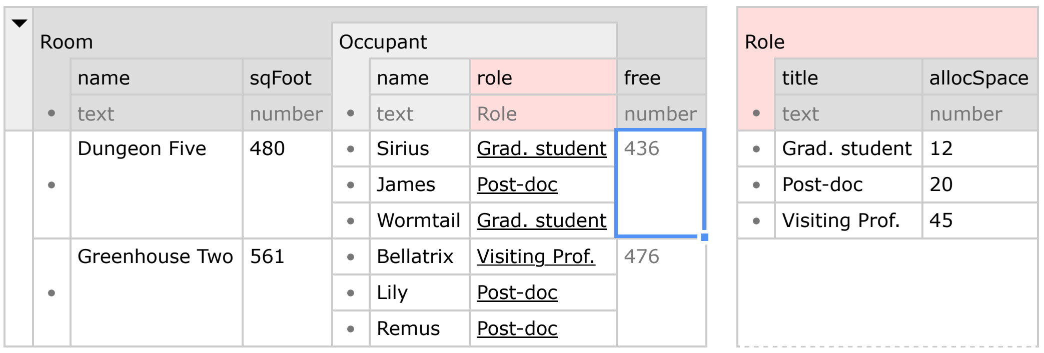
	A	B	C	D	E	F
1	room	sq. footage	occupant	role	alloc.	free
2	Dungeon Five	480	Sirius	Grad. student	12	**436**
3			James	Post-doc	20	
4			Wormtail	Grad. student	12	
5	Greenhouse Two	561	Bellatrix	Visiting Prof.	45	**476**
6			Lily	Post-doc	20	
7			Remus	Post-doc	20	
8	role	alloc. space				
9	Grad. student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

=VLOOKUP(D5, A$9:B$11, 2)
=B2−sum(E2:E4)
=B5−sum(E5:E7)

Figure 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.

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. 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 put 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



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 $\widehat{=}$ sqFoot − sum[v : Occupant](v.role.allocSpace)

Figure 3 An Object Spreadsheet for the space allocation example.

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 with per-cell formulas as the fundamental data model in favor of one that directly supports nested sets of objects. In our model, every formula defines a computed field of an object type and is automatically evaluated on each object of that type in the sheet, ensuring uniformity; a root object is available to hold global values and formulas. We visualize the data using 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 [7, 8, 19]. The result is shown in Fig. 3.

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. 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 admin 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 of the visualization 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. 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.

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 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 the 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.

Mutations. Finally, the developer must specify the kinds of mutations that users may make to the application's state. Assume that the admin from the previous examples 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 PostgreSQL does.

However, some of our target applications, such as Got Milk (see Section 5), 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 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
```

2 Data Model

Our data model is visually similar to a standard spreadsheet (and indeed we shall refer informally to a particular instance of the data model as a "spreadsheet"), but also derives from the entity-relationship (ER) model proposed as a precursor to the design of relational schemas. It differs from a pure relational model in several ways that we believe will make it a better match to the developer's mental model of their application. It offers:

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

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 (details in Section 3.3); cells in object columns (*object cells*) do not store data but instead represent distinct *object identities*, visualized as bullets. To illustrate this, Fig. 4 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 not shown. (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.

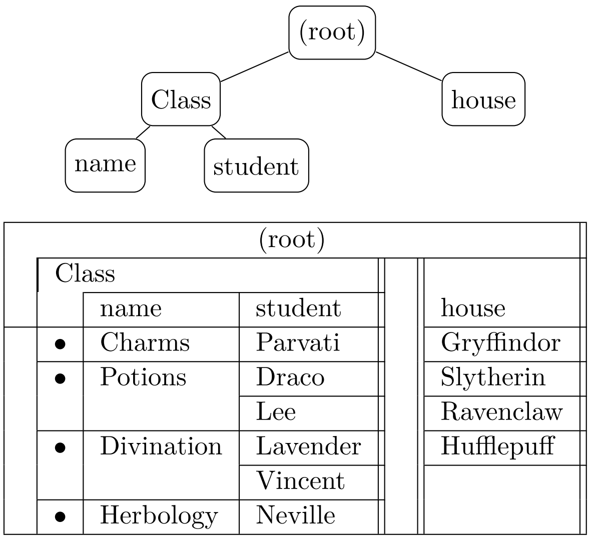
In the example of Fig. 4, "Class" has two child columns, "name" and "student". "Class" and "house" are both directly descended from the root column (not shown in the figure). Fig. 4 presents the columns of this spreadsheet as a tree, expressing the ownership relations. To indicate the column hierarchy 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. 5 shows the tree of cells for the sheet of Fig. 4, 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,[1] 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 using the normal spreadsheet options (e.g., lexicographic order for strings, chronological order for

[1] 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.



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

Figure 4 A simple spreadsheet with nested elements and its column hierarchy.

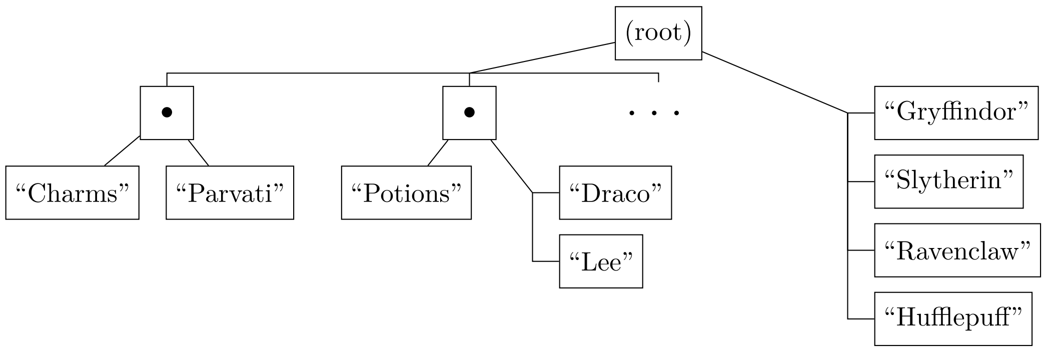


Figure 5 Cell hierarchy (only some of the cells are shown). Forked edges indicate cell families.

dates)[2]. [3]

The most basic use case of a spreadsheet is data entry. The owner of a spreadsheet, 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. Section 3 describes these computations and their semantics.

[2] This feature has not yet been implemented.
[3] To simplify the example in Fig. 4, 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.

2.1 Formal Specification

The object spreadsheet data model has two main parts: a *data schema*, which contains the definition of the column hierarchy; and a *data instance*, which holds actual cells and their values.

2.1.1 Data Schema

Our system provides a set of primitive types $\mathbb{P}$, including "text", "number", "boolean", etc. A function $\mathcal{V}$ is initially defined on each of these types, giving the set of values of that type.

▶ **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.
- A *root column* $R \in \mathbb{OC}$ and a *parent function* $\mathsf{p} : \mathbb{C} \setminus \{R\} \to \mathbb{OC}$ 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 $\mathbb{C}^+ = \mathbb{C} \setminus \{R\}$, $\mathbb{OC}^+ = \mathbb{OC} \setminus \{R\}$.
- A function $\mathsf{name} : \mathbb{C}^+ \to$ string giving a name to each non-root column.
- A type assignment $\Gamma : \mathbb{C} \to \mathbb{T}$ (where $\mathbb{T} = \mathbb{P} \cup \mathbb{OC}$) such that $\Gamma[C] = C$ for every $C \in \mathbb{OC}$.

With this notation, each object column also functions as a type whose values are *references* to any objects that could possibly exist in the column.

2.1.2 Data Instance

▶ **Definition 2.** An *instance* M of a schema Σ consists of:

- A set $\mathbb{E}$ of *cells*.
- A *root cell* r and a function $\mathsf{col} : \mathbb{E} \to \mathbb{C}$ assigning cells to columns, such that $\mathsf{col}^{-1}[R] = \{r\}$. We write $e : C$ to mean $\mathsf{col}[e] = C$, and denote $\mathbb{E}^+ = \mathbb{E} \setminus \{r\}$.
- A parent cell function $\mathsf{p} : \mathbb{E}^+ \to \mathbb{E}$. As with columns, we use $d \rightsquigarrow e$ to say $\mathsf{p}[e] = d$.
- A value assignment val, which maps each cell $e \in \mathbb{E}$ to a value $v : \Gamma[\mathsf{col}[e]]$, where $v : \mathcal{T}$ is defined as follows:
  - For a primitive type $\mathcal{T} \in \mathbb{P}$, $v : \mathcal{T}$ just means $v \in \mathcal{V}[\mathcal{T}]$.
  - For an object column $C \in \mathbb{OC}$, $v : C$ is as defined above, i.e. $v \in \mathbb{E}$ and $\mathsf{col}[v] = C$.

An instance must satisfy the following properties:

- Homomorphism: the parent-child relationship between cells respects that of columns. That is, if $d \rightsquigarrow e$, then $\mathsf{col}[d] \rightsquigarrow \mathsf{col}[e]$.
- Value uniqueness: given a cell $d : D$ and a child column $D \rightsquigarrow E$, the values of d's children in E comprise a logical set, so there are no repeating values, that is —

$$\forall e_1, e_2 : E.\;\; d \rightsquigarrow e_1 \;\wedge\; d \rightsquigarrow e_2 \;\wedge\; e_1 \neq e_2 \implies \mathsf{val}[e_1] \neq \mathsf{val}[e_2]$$

- Object identities: the value of an object cell is itself, i.e., $\mathsf{val}[e] = e$ for $e : C, C \in \mathbb{OC}$.

The parent function partitions the cells in each column. We call these partitions *families*. A family is formally addressed by a pair $\langle C, d\rangle$ where $C \in \mathbb{C}^+$ and $d : \mathsf{p}(C)$ and contains all cells in C whose parent is d.

Value columns having a type that is another object column, $\Gamma[C] \in \mathbb{OC}$, are used to create *cross references*. Cells in these columns hold references to other cells in the respective object columns, and play a similar role to that of references in object-oriented languages. Cross references are discussed in more detail later in the paper (Section 3.3), after we have defined the formula language.

```
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
```

Table 1 Formula syntax.

3 Formulas and Computation

Formulas are assigned to columns rather than to individual cells as 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 will 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 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, one can chain navigations.)

For simplicity of design, all formula expressions evaluate to sets, following the style of Alloy [22]. Navigation is done using the *dot notation*: *cell*.*ancestor* or *cell*.*child*, and is naturally lifted to sets by taking the union of the results of the navigation on each element. The column name 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 perform this resolution, we must know the type of *cell*, so 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 Table 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 syntax for `sum` resembling the mathematical $\sum$ notation: `sum[p:Part](p.price)`.[4]

When a formula is evaluated in the context of a family, an implicit variable `this` is bound to the parent cell of the family. 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 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).

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

3.1 Formula Semantics

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 determined automatically during each type-checking pass. This process may fail if there is a cyclic dependency among columns of unspecified type, in which case the developer can break the cycle by specifying one of the types. For as long as a column's formula is ill-typed or fails to match a specified type, evaluation of cell values in the column is postponed.

The formal type-checking rules are listed in Table 2. Formulas are checked in a type environment $\Gamma = \{\mathtt{this} \mapsto C\}$ for the appropriate parent column C. In particular, (LU-$\downarrow$) and (LU-$\uparrow$) describe the lookup rules for down/up navigation; $\mathsf{name}[C]$ denotes the name assigned to C in Σ. In addition, it should be pointed out that navigation must be unambiguous. That is, if two distinct judgments $\mathrm{lu}(C.id) = lu_1$, $\mathrm{lu}(C.id) = lu_2$ (with $lu_1 \neq lu_2$) can be obtained for the same expression, the type checker rejects the formula and reports an ambiguity error.

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.

[4] We would like to support the simpler syntax `sum(Part.price)`, but under the current model, `Part.price` is a set without duplicates. Introducing multiset-valued expressions would be one way to solve this problem.

$$\frac{}{\Gamma, v : T \vdash v : T}\,(\text{VAR})$$

$$\frac{C \rightsquigarrow D \quad \mathsf{name}\big[D\big] = id}{\text{lu}(C.id) = \langle D, \downarrow\rangle}\,(\text{LU}-\downarrow) \qquad \frac{B \rightsquigarrow^{*} C \quad \mathsf{name}\big[B\big] = id}{\text{lu}(C.id) = \langle B, \uparrow\rangle}\,(\text{LU}-\uparrow)$$

$$\frac{\Gamma \vdash e : C \quad \text{lu}(C.id) = \langle D, d\rangle \quad {\scriptstyle (d \in \{\uparrow,\downarrow\})}}{\Gamma \vdash e\,\texttt{.}\,id : \Gamma[D]}\,(\text{NAV}_{(.)})$$

$$\frac{\Gamma \vdash \texttt{this}\,\texttt{.}\,id : T}{\Gamma \vdash id : T}\,(\text{NAV}_{\text{this}})$$

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

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

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

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

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

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

Table 2 Formula type-checking rules.

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.

The evaluation of formulas is given via denotational semantics in Table 3. In the context of an instance M, the semantic function $[\![\,]\!]$ maps a formula expression e and an assignment σ to a set of values of type T, assuming that σ maps the variables to set types respecting Γ and that $\Gamma \vdash e : T$. The notation $\rightsquigarrow^{*}$, used in both typing rules and semantics, is the reflexive-transitive closure of the respective $\rightsquigarrow$ relation on columns/cells.

3.2 Computational Model

The semantics describes the steps of the computation performed, but for completeness we need to explain the computation of the entire spreadsheet, and in particular that cyclic references and formulas yield undefined results, and that a spreadsheet can be extended by the addition of formulas in the expected way.

The basic unit of computation for formulas is the *family*. Recall that a family is formally addressed by a pair $\langle C, d\rangle$ where $C \in \mathbb{C}^{+}$ and $d : \mathsf{p}(C)$.

▶ **Definition 3.**

$$\begin{aligned}
&[\![v]\!]\sigma = \sigma v && \text{if } v \text{ is a variable}\\
&[\![id]\!]\sigma = [\![\texttt{this.}id]\!] && \text{if } id \text{ is a column name}\\
&[\![e\,\texttt{.}\,id]\!]\sigma = \begin{cases}\{\mathsf{val}[c] \mid \alpha \in [\![e]\!]\sigma, c : C, c \rightsquigarrow^* \alpha\} & \text{if } \Gamma \vdash e : D, \mathrm{lu}(D.id) = \langle C, \uparrow\rangle\\ \{\mathsf{val}[c] \mid \alpha \in [\![e]\!]\sigma, c : C, \alpha \rightsquigarrow c\} & \text{if } \Gamma \vdash e : D, \mathrm{lu}(D.id) = \langle C, \downarrow\rangle\end{cases}\\
&[\![l]\!]\sigma = \{l\} && \text{if } l \text{ is a primitive literal}\\
&[\![\$]\!]\sigma = \{r\}\\
&[\![\texttt{\{}\, e_1 \texttt{,} \ldots \texttt{,} e_n \,\texttt{\}}]\!]\sigma = \textstyle\bigcup_i [\![e_i]\!]\sigma\\
&[\![e_1 = e_2]\!]\sigma = \big\{\big([\![e_1]\!]\sigma = [\![e_2]\!]\sigma\big)\big\} \qquad [\![e_1 \;\texttt{in}\; e_2]\!]\sigma = \big\{\big([\![e_1]\!]\sigma \subseteq [\![e_2]\!]\sigma\big)\big\}\\
&[\![e_1 + e_2]\!]\sigma = \{x + y\} && \text{if } [\![e_1]\!]\sigma = \{x\}, [\![e_2]\!]\sigma = \{y\}\\
&[\![f\,\texttt{(}e_1\texttt{,} \cdots \texttt{,} e_n\texttt{)}]\!]\sigma = f\,([\![e_1]\!]\sigma, \cdots, [\![e_n]\!]\sigma)\\
&[\![\texttt{\{}v\texttt{:}e\texttt{|}c\texttt{\}}]\!]\sigma = \big\{\alpha \in [\![e]\!]\sigma \;\big|\; [\![c]\!](\sigma[v \mapsto \alpha]) = \{true\}\big\}\\
&[\![\texttt{sum[}v\texttt{:}e\texttt{]}\;\texttt{(}a\texttt{)}]\!]\sigma = \textstyle\sum_{\alpha \in [\![e]\!]\sigma} [\![a]\!](\sigma[v \mapsto \alpha])
\end{aligned}$$

Table 3 Denotational semantics for formulas.

1. A *computed instance* M of a schema Σ has all the components of an instance (Definition 2), plus a set $\mathbb{F} \subseteq \mathbb{C} \times \mathbb{E}$ of *computed families*.
2. A family $\langle C, d\rangle \in \mathbb{F}$ in a computed instance M *respects* the formula φ when:
   - If C is a value column:
     $$\{this : p(C)\} \vdash \varphi : \Gamma[C] \quad \text{and} \quad [\![\varphi]\!]\{this \mapsto d\} = \{\mathsf{val}[e] \mid e : C, d \rightsquigarrow e\}$$
   - If C is an object column, where K is its key column ($C \rightsquigarrow K$):
     $$\{this : p(C)\} \vdash \varphi : \Gamma[K] \quad \text{and} \quad [\![\varphi]\!]\{this \mapsto d\} = \{\mathsf{val}[k] \mid e : C, k : K, d \rightsquigarrow e \rightsquigarrow k\}$$

   ($d \rightsquigarrow e \rightsquigarrow k$ abbreviates $d \rightsquigarrow e$ and $e \rightsquigarrow k$.)
3. A *program* $\Phi : \mathbb{C}^+ \rightharpoonup \text{expr}$, $\kappa : \mathbb{OC}^+ \rightharpoonup \mathbb{VC}$ is an assignment of formulas to a set of columns of the schema known as the *computed columns*. Computed object columns are assigned respective key columns such that $C \rightsquigarrow \kappa[C]$. The remaining columns are known as *state columns*. Programs must have the property that no state column is the child of a computed column; this is necessary for state instances as defined below to exist. Families in state columns and computed columns are called *state families* and *computed families*, respectively.
4. M *respects* Φ, κ if for every $C \in \mathrm{dom}[\Phi]$ and $e : C$, M has $\langle C, \mathsf{p}[e]\rangle \in \mathbb{F}$, and every computed family $\langle C, d\rangle \in \mathbb{F}$ respects $\Phi[C]$ (with key $\kappa[C]$ where appropriate).

▶ **Definition 4.**
1. An *input instance* is a computed instance with finitely many cells and with $\mathbb{F} = \varnothing$.
2. In an instance M that respects a program Φ, κ, a computed family $\langle C, d\rangle$ *depends* on a family $\langle C', d'\rangle$ if evaluation of $[\![\Phi[C]]\!]\{this \mapsto d\}$ attempts to read from cells belonging to $\langle C', d'\rangle$.
3. M is *good* for Φ, κ if it respects Φ, κ and the dependency relation on computed families contains no infinite forward chains (and in particular, no cycles).
4. An instance M' is an *extension* of an instance M for the same schema if the sets $\mathbb{E}$ and $\mathbb{F}$ of M' are supersets of those of M, the functions col and val of M' are extensions of those of M, and M' does not add any new cells to families that exist in M.
5. A *spreadsheet* consists of a schema Σ, a program Φ, κ for Σ, and an input instance M that respects Φ, κ.

The following proposition is proved in the appendix:

▶ Proposition 1. In any spreadsheet, there is a good extension $\widehat{M}$ of M that is an extension of any other good extension of M.

We call $\widehat{M}$ as defined in Proposition 1 the *computed extension* of M. It contains all families that can be evaluated successfully. All readers of the data model, including the spreadsheet UI and web application views, use the computed extension.[5]

It is impossible to write a formula that returns an infinite set, and consequently, the computed extension always contains finitely many cells and can be computed in its entirety in finite time. The computational model in this section works for such infinite spreadsheets, though the spreadsheet UI would need to offer controls to show only parts of the sheet at a time and the runtime system would need to compute those parts on demand. These capabilities would be important for large finite sheets as well, such as those that back views, described in the next section.

3.3 Cross References

As explained in Section 2.1.2, some cells contain references to other (object) cells. References can be followed directly, unlike foreign keys in a relational database, which require an explicit join or subquery on the other table. References are first-class values; their closest analogues in a traditional spreadsheet are cell reference strings such as "A11" that can be treated as data and later passed to a dereferencing function such as INDIRECT.

These references can be obtained in two ways:

- Computed by a formula, as described in the previous subsections. A formula evaluation produces a set of values, which can be references if the type of the formula resolves to some object column.
- Entered by the user.

While the first case has been thoroughly covered, the second case has some subtle issues; in particular, the user-entered references need to be stored with the state data (for computed values, we assume they can be recomputed when needed). A form of addressing is needed that is persistent and stable as data changes, especially in the presence of computed data.

We apply separate treatment for references to cells in state columns and to computed columns. For state object cells, we assume we have an infinite set $\mathcal{T}$ of *tokens*, and a unique token is allocated to every new state object cell when it is created. We then store a bijective mapping $tok : \mathbb{E}^+ \rightharpoonup \mathcal{T}$ with the serialized state data, and store any reference to a state cell e as $tok^{-1}[e]$. This resembles the practice of assigning unique identifiers to rows in a relational database. It should be noted, however, that the tokens are encapsulated and hidden from the user in the same way that addresses are encapsulated in Java, for example.

Our implementation also supports references from state cells to computed objects, but since this situation is quite rare, we describe this feature as an extension and defer it to Section 7.

For an example, Fig. 3 in the introduction shows a spreadsheet containing references. The column “role” of the object “Occupant” contains references to objects of type “Role”. Reference values are marked with an underline, and the text defaults to the first field of the referenced object.

[5] Our formalization suggests that the tool can simply fail in the presence of errors such as cycles, but in practice of course it should provide feedback about the location of the error, and show any intermediate results that were computed up to that point.

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

Hello, Molly.

Meetings for Fred	**selected**	**pick one:**
	Snape @ 1pm	$\boxed{\circ}$ Trelawney @ 4pm
	$\boxed{\times}$	$\boxed{\circ}$ Trelawney @ 5pm
		$\boxed{\circ}$ Trelawney @ 6pm
Meetings for George	**pick one:**	
	$\boxed{\circ}$ Flitwick @ 10am	
	$\boxed{\circ}$ Flitwick @ 11am	

Figure 6 A view model and one instance for scheduling parent-teacher meetings, with an example rendering.

Certain UI features can make references more manageable: for example, highlighting the target of a reference when mousing over its containing cell, a "jump to" operation that moves the spreadsheet cursor there, and the ability to choose a "display field" other than the first field, which can also be a computed field.

3.4 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 regular 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 is shown in Fig. 6. 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"$\rightsquigarrow$"Child" and "Child"$\rightsquigarrow$"Meeting", matching the containment structure on the HTML form below.

To understand how the control aspect works, notice the buttons $\boxed{\times}$ and $\boxed{\circ}$ 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 section; the important thing to notice is

```
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
```

Table 4 Procedural language syntax.

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.

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 [27]. 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.

In keeping with the spreadsheet philosophy, transaction procedures are written in a simple procedural language that does not include all the abstraction capabilities of general-purpose programming languages. The grammar is shown in Table 4. 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. A formal definition of mutations is shown in Appendix A.2.

`check` 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 the user 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. `let` bindings extend Γ with more type assignments according to the types of expressions, which are formulas and therefore follow the same rules of Table 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, 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.

Clearly, writing a transaction procedure presents a greater challenge to an end-user developer than writing formulas. For create, update, and delete 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 [17]. 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.

5 Experiments and Evaluation

We have built a prototype of the object spreadsheets execution engine and the developer user interface on top of the Meteor web framework [26]; all our applications thus inherit the reactivity of Meteor. The developer UI is rendered via a Handsontable [20] 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 currently be edited in the user interface (so they must be provided in a file).

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 templates provided by Meteor (eventually, template creation 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 schedule meetings only with teachers that teach one of their children; they can only schedule one meeting per Enrollment; and they cannot book an occupied time slot.
- 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 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.

Case study. 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. 7 shows a sample sheet containing some concrete data. Column names, their types, and their hierarchy are shown by the header of the spreadsheet. The 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. 7, 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 slightly unusual representation, 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.

5.1 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

	Data		Formulas	Procedures
	# 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 Sizes of sample applications.

	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

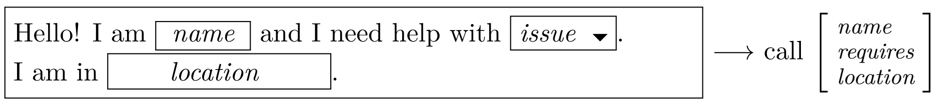
```
call.assign := user
```

forfeit call : Call

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

done call : Call

```
delete call
```


Hello! I am *name* and I need help with *issue*.
I am in *location*.

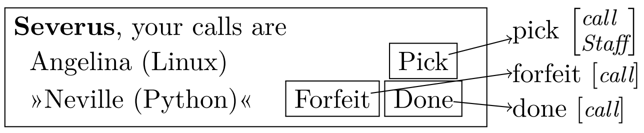

Severus, your calls are
Angelina (Linux)
»Neville (Python)«

Figure 7 Data model, formulas, transaction procedure code, and HTML forms associated with the simple queuing example "Hack-q" in the case study.

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 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.

6 Related Work

Spreadsheet-backed application builders. The clearest example of a tool for building spreadsheet-backed web applications with their own permanent state is Quilt [9]. 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 bound 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, probably because these applications do not need to maintain their own richly structured state across programmatic updates, which is challenging for traditional spreadsheets. Gneiss [11, 12] and SpreadMash [23] 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 [16] 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 how the UI will scale to development of our target applications.

App2You [24] and AppForge [34] 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 [24] does not state the form of filters supported by App2You, though it mentions a formula language as a future extension.

Finally, 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 [30]. 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 [7] 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 [8, 6], a tool for exploring existing relational databases. Because of the close similarity of the work, we asked the authors for details about the system that are not yet published. 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 [25] 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 [19] 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 recording them in the script.

Spreadsheets with declarative repetition of formulas. MDSheet [15] lets a developer define the structure and formulas of a spreadsheet using the ClassSheets [18] modeling format, which supports repeating groups of rows and columns and a dot notation for navigation, and then maintains it as the developer adds and removes repetitions. However, this system supports only one level of repetition along each axis of the grid, which limits its expressive power.

The Sumwise [28, 29] spreadsheet tool lets developers annotate rows and columns based on their roles in the sheet and then declaratively bind formulas to groups of cells based on those annotations. However, the available documentation is insufficient for us to evaluate its expressive power compared to that of Object Spreadsheets, and it does not address programmatic mutations.

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 [35] 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 [10] 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 [32] and the spreadsheet of Clack and Braine [14] 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 these systems do not provide any help working with variable-size data in the spreadsheet grid.

Other tools that bridge databases and spreadsheets. Senbazuru [13] 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. [21] 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.

7 Extensions

UI builder. We envision a UI builder 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.

References to computed objects. As described in Section 3.3, references to state objects are done through tokens. For objects created using a formula, this approach would not work since objects are recreated when the formulas are reevaluated, and the tokens will not be stable. Instead of devising a consistent token-allocation strategy for computed objects, we take advantage of the existing compute engine to reduce *state references to computed objects* into *computed references*: we create value columns corresponding to the *key* of the computed object and its *parent*. Then we add a formula that pulls the right object from the parent and places it in a third column. This technique can be extended to more than one key, in case computed objects are nested.

Access control and privacy. Portions of the application's read access control policy can be factored out into their own computed columns and then referenced from the formulas for each view. For example, the developer could define a column giving the set of domain objects that each user is allowed to see and then define each view to be limited to the objects listed in that column for the current user. The developer can even opt to define a per-user filtered copy of the entire application state and base all views on this copy, achieving a similar effect to the centralized read access control policies of Sunny [27].

Bridge to other services. Naturally, many web applications require intercommunication with web services such as e-mail, social media, news feeds, etc. A mechanism for hooking spreadsheet applications with external sources can be implemented by designating a table that gets filled by a service "robot" instead of by the user; for example, a table connected to an e-mail account, where a row is inserted whenever a message is received. Spreadsheet formulas can then operate on the data in the table to do any filtering or aggregation required. Similarly, tables can be used as sinks, where the insertion of a row (typically by a transaction) will trigger a message to be sent.

8 Conclusion

Our experience with the data model and the computational model shows that they yield compact programs that are easy to understand and easy to change, making them suitable for rapid prototyping without up-front design and for situations where the requirement specifications tend to change frequently. We have shown that spreadsheets can greatly benefit from having more structure built into them, without making it harder to write formulas, and while keeping the data model coherent with the visual representation. Programming with formulas can be made even easier with additional point-and-click support, which reduces the need to type identifier names and is very intuitive for spreadsheet users. Having a statically typed language aids in early detection of mistakes, but does not burden the developer with annotations, since most of the types can be inferred. Tighter integration with a UI builder can help filling in the types of transaction parameters by selecting them from the view. Most importantly, the application is always "live", with the view reflecting the current value of the state and the formulas that have been entered, this breaking the code-build-debug cycle that slows down and obscures conventional programming.

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

References

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

6 The SIEUFERD project. http://people.csail.mit.edu/ebakke/sieuferd/index.html.

7 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.

8 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.

9 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.

10 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.

11 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.

12 K.S.-P. Chang and B.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.

13 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.

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

15 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.

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

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

18 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.

19 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.

20 A minimalist excel-like data grid editor for html & javascript. www.handsontable.com.

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

22 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.

23 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.

24 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.

25 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.

26 An open source platform for building web applications. `www.meteor.com`.

27 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.

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

29 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.

30 Richard Pawson. *Naked objects*. PhD thesis, Trinity College, 6 2004.

31 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.

32 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.

33 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.

34 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.

35 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.

A Appendix

A.1 Proof of existence of the computed extension

Proposition 1 *In any spreadsheet, there is a good extension $\widehat{M}$ of M that is an extension of any other good extension of M.*

Proof. First we show that no two good extensions M_1, M_2 of M can have different data in the same computed family. Suppose for the sake of contradiction that such M_1, M_2 exist, and let S be the set of families that appear in both M_1 and M_2 with different data. Since the family dependency relation in M_1 has no infinite forward chains, we can walk dependencies to find a family $f \in S$, none of whose dependencies in M_1 are in S. So either all of these dependencies are present in M_2 with the same data, in which case the formula semantics give the same data for f in M_2, or at least one is absent, in which case the formula semantics do not give a result for f in M_2 at all. This contradicts the choice of $f \in S$.

Now, we claim that the union of all good extensions of M is the desired $\widehat{M}$. One can show that it is a well-formed instance, and it is clearly an extension of each of these extensions. It remains to show that $\widehat{M}$ is good. Each computed family f in $\widehat{M}$ comes from some good extension M_1 of M (in general, not unique), and all of the dependencies of f in M_1 appear in $\widehat{M}$ with the same data, so the formula semantics give the same data for f in $\widehat{M}$, meaning that f respects its formula in $\widehat{M}$. And if there were an infinite dependency chain from f in $\widehat{M}$, the same chain would exist in M_1, contrary to the assumption that M_1 is good. ◀

A.2 Procedure Semantics

The purpose of procedural code embedded in a spreadsheet is to alter its state instance M, that is, to add and remove cells in state columns. Each transaction in progress has a local variable assignment σ that is initialized with the parameters of the transaction procedure and can be updated by `let` bindings. All formulas in a statement are evaluated with respect to σ and the computed extension $\widehat{M}$ of M, as it exists at the time the statement begins.

The semantics of the statements that mutate state are given in Table 6. $\Delta^- \subseteq \mathbb{E}$ is the set of cells to delete from M. Δ^+ is a set of tuples of one of the following forms:

- $\langle\langle C, d\rangle, v\rangle$ where $\langle C, d\rangle \in \mathbb{F}$, $C \in \mathbb{VC}$ and $v \in \mathcal{V}(\Gamma[C])$, meaning that a cell of value v should be created in family $\langle C, d\rangle$;
- $\langle\langle C, d\rangle, k\rangle$ where $\langle C, d\rangle \in \mathbb{F}$, $C \in \mathbb{OC}'$ and $k \in \mathcal{V}(K_\Sigma[C])$, meaning that a cell of key k should be created in family $\langle C, d\rangle$.

When a cell is created or deleted, its child families in all child state columns are implicitly added or removed. Key families and the values of object cells are implicitly updated as prescribed by the definition of a data instance. For suggestiveness, we notate a tuple $\langle\langle C, d\rangle, x\rangle \in \Delta^+$ as $d \rightsquigarrow \boxed{x} : C$.

The first four statements in Table 6 operate on a set of families addressed by the expression $e_1.id$, where e_1 determines the parent cell of each family, and id identifies the child column where the family lives (only down-navigation can be used address families). For these statements, we define the *target columns* C_1 and C_2 as the parent and child columns, respectively, given by $\Gamma \vdash e_1 : C_1$ and $\langle C_2, \downarrow\rangle = \mathrm{lu}(C_1, id)$.

$\boxed{e_1\,\mathtt{.}\,id\ \mathtt{:=}\ e_2}$

$\Delta^- = \{e \mid d \rightsquigarrow e, e : C_2, d \in [\![e_1]\!]\sigma\}$

$\Delta^+ = \left\{d \rightsquigarrow \boxed{\beta} : C_2 \;\middle|\; d \in [\![e_1]\!]\sigma, \beta \in [\![e_2]\!]\sigma\right\}$

$\boxed{\mathtt{to\ set}\ e_1\,\mathtt{.}\,id\ \mathtt{add}\ e_2}$

$\Delta^- = \varnothing$

$\Delta^+ = \left\{d \rightsquigarrow \boxed{\beta} : C_2 \;\middle|\; d \in [\![e_1]\!]\sigma, \beta \in [\![e_2]\!]\sigma \setminus \{\mathsf{val}[e] \mid d \rightsquigarrow e, e : C_2\}\right\}$

$\boxed{\mathtt{from\ set}\ e_1\,\mathtt{.}\,id\ \mathtt{remove}\ e_2}$

$\Delta^- = \{e \mid d \rightsquigarrow e, e : C_2, d \in [\![e_1]\!]\sigma, \mathsf{val}[e] \in [\![e_2]\!]\sigma\}$

$\Delta^+ = \varnothing$

$\boxed{\mathtt{new}\ e_1\,\mathtt{.}\,id}$

$\Delta^- = \varnothing$

$\Delta^+ = \left\{d \rightsquigarrow \boxed{\text{fresh}\ \mathbb{E}} : C_2 \;\middle|\; d \in [\![e_1]\!]\sigma\right\}$

$\boxed{\mathtt{delete}\ e_1}$

$\Delta^- = \left\{e' \;\middle|\; e \in [\![e_1]\!]\sigma, e \rightsquigarrow^* e'\right\}$

$\Delta^+ = \varnothing$

Table 6 Semantics for mutating statements in terms of Δ^- and Δ^+. S and C_2 are defined in the text.