Abacus: A New Spreadsheet Paradigm for Reducing Errors

Steve Chapel
schapel@umich.edu

Rahul Jha
rahuljha@umich.edu

ABSTRACT
When spreadsheets were initially developed, computers had low-resolution screens which could hold very little information and display only text-based information. Today, although nearly every computer has a large, high-resolution color graphical display, we are stuck with the paradigm of spreadsheets as a huge array of cells in which formulas are copied and modified. Formulas cannot be seen except for a cell-at-a-time view. Cells are referred to with an arcane letter-and-number syntax that belies the relative nature of the relationship between cell names and their use. This paper explores several ideas to form a new paradigm for spreadsheets for the purpose of making them easier to use correctly.

Categories and Subject Descriptors
D.1.7 [Software]: Programming Techniques—Visual Programming; H.4.1 [Information Systems Applications]: Office Automation—Spreadsheets

General Terms
Reliability

Keywords
visual programming, spreadsheets, units

1. INTRODUCTION
Spreadsheets are a popular technology for handling financial calculations and other tasks. The spreadsheets provide a programming environment that is very close to the functional programming paradigm. As such, it has received significant interest from the programming languages community. [1] explores the spreadsheet paradigm in great detail and present a very sophisticated programming environment built around the spreadsheet model. [6] and [5] explore functional abstraction and modularity in spreadsheets. Other work has focused on reducing spreadsheet errors ([2] [4]) using different techniques. Using types for units has also been explored by [3].

In Abacus, we wanted to bring together some of these ideas together into a functional implementation. The main contributions of this work are:

- A visual framework for declaring functions within spreadsheets and linking them to variables and columns.
- A dynamic type checking system that detects and reports unit-based errors.

The first of these focuses on allowing users to have functional abstraction within their programs by declaring a function once and then being able to use it on several sets of data. We have focused on providing a clear separation between the data and code sections of a spreadsheet. We also provide a mechanism for testing user functions before they are applied to large sets of data.

The second one focuses on ensuring that common spreadsheet errors are detected and reported appropriately. The complete system is described in section 3. The rest of the paper is organized as follows. Section 2 introduces the most common spreadsheet errors that we are trying to deal with. Section 3 introduces our language for spreadsheet programming and also explains the type system for units. We discuss the implementation of Abacus in Section 4. Some related work is presented in Section 5. Section 6 concludes with some pointers to future work.

2. SPREADSHEET ERRORS
The following are the common types of errors that occur in spreadsheets. We address each of these sources of error in turn.

2.1 Physical Coupling of Unrelated Cells
Spreadsheets typically contain a vast array of cells, for more than most spreadsheet authors will use. Usually, this vast array is partitioned into logically separate areas that are nonetheless physically connected. If two logically separate tables are side-by-side and the user adds a row to the left table, a row is mistakenly added to the right table as well. Our method for reducing this type of error is to create tables of cells only as big as needed, with each physically separate table holding only one logical entity. With this approach,
the user can add or delete a row or column without affecting other logically separate tables, which may not be displayed onscreen during the operation. See Figure 1 for an example of this problem with traditional spreadsheets.

2.2 Cut-and-Paste Programming
When using formulas in traditional spreadsheets, formulas are written in one cell, then copied to other cells. When the user wishes to modify the formula, the user must remember to manually copy the formula to all the cells it was copied to before.

A problem related to cut-and-paste programming occurs when the user copies a formula to all cells in a column and later adds a row in the middle of the column. The formula for the cell just under the new row is altered so that it skips the inserted row. When the user fills in values for that row, they are skipped. Our system helps eliminate this type of error by assigning the same formula to a block of cells. When a new row is added to the block of cells, the cells in the row have the same formula, with the same relationship between cells. To help assist the user to visualize which cells have the same formula, all the cells that have the same formula are highlighted when the user selects one of the cells that contains the formula. See Figure 2 for an example of this problem with traditional spreadsheets.

2.3 Independent Testing
By writing each function separately from the data it is eventually intended to operate on, we can allow the user to test the function with test data before using it on the intended data.

2.4 Visual Programming
By placing formulas in a separate area of the spreadsheet, we reduce the formula of an output cell to the form of a function call that uses cells relative to the output cell as inputs and the cell itself as the output of the function. We can then visually represent this relationship between inputs to a function rather than presenting it as a text formula that may be hard to read.

2.5 Uninitialized Cells
Spreadsheets typically treat a cell that does not have any information in it as if it has the numerical value of zero. This can cause errors to occur silently. By treating cells with no entered data as an uninitialized value, we can alert the user to missing values rather than silently assigning a default value to them such as zero for a numeric field or the empty string for a string field.

2.6 Unit Errors
In traditional spreadsheets, numerical values are treated as raw numbers without any physical interpretation. In Abacus, values have unit types, and the user receives a type error on incompatible type operations. Allowed formats are determined by type, so a value in meters cannot be displayed with a monetary format.

3. LANGUAGE
3.1 Syntax and Semantics
Our language is essentially a subset of JavaScript that includes the following features:

Numeric expressions: + - * / %
Comparisons: > >= < <= == !=
Logical expressions: && || !
Structured types: arrays with indexing (e.g. arr[1]), objects with fixed field names (e.g. obj.field).
Statements: assignment, if, if-else, while, return
Function declarations and calls

Objects with variable field names are disallowed so we can do static type checking, i.e. we can know the type of each field, which we may not be able to do if we allow variables as field names. Arrays with variable indices can be typechecked because they are heterogeneous.

The built-in functions include:

floor Returns the largest integer not greater than the argument
length Returns the length of an array
raw Returns the dimensionless value of the argument, i.e. without units
unit Returns an array describing the units of the argument
one Returns the value 1 with the units described in the array argument

The language is a functional language in that all expressions are statements, statements evaluate to values, and there are no global variables (all variables are implicitly local) except for functions. Unlike many functional languages, however, statements are not allowed inside expressions, and the side effect of assigning to local variables is allowed.

Units are type checked at runtime. They are checked for equality on addition, subtraction, and comparing. For multiplication and division, units are multiplied and divided, respectively. The remainder has the same units as the dividend so that the following equation is always satisfied:
\( \text{floor}(a / b) + (a \% b) = a \)

3.2 Typing
A simple example demonstrates that performing static type checking on our code to ensure units of measurement are not mixed inappropriately, may hinder the user from writing many useful programs. Consider the following function which computes a nonnegative integer power of an arbitrary floating point number:

```javascript
function power(x, p) {
  i = 1;
  result = 1;
  while (i <= p) {
    i = i + 1;
    result = result * x;
  }
  return result;
}
```

If a static type checker attempts to determine the run time units of measurement of \( \text{result} \), it will be unable to do so unless \( x \) is a dimensionless number. Rather than checking units of measurement statically, we can compute the units during run time and alert the user when the value computed by a function has units that are incompatible with the units of the cell the value is being stored to.

The language has a syntax for annotating constants in program text with units of measurement, e.g. 9.8 (newton-meter), 15 (pounds/in^2), 440 second^{-1}. The number zero can be handled in two distinct ways. Abacus always assigns a unit to zero so that it is type checked during operations. Kennedy’s system of units[3] considers zero as a value of any unit. Handling zero as a polymorphic value makes sense for units in which 0 of one unit is the same as 0 of a compatible unit (e.g. 0 meter = 0 feet) but would not make sense for degrees Celsius and degrees Fahrenheit, where 0 denotes different temperatures depending on the unit used for degrees.

In some cases it is necessary to allow the user to cast an expression to a different unit of measurement at run time. For example, if the user wished to compute \( \sqrt{4 \text{ meter}^2} \) and \( \sqrt{9 \text{ foot}^2} \), the units of the result would need to be computed at run time.

We allow the user to get the raw value of a type in dimensionless units, provide a way to get the units of a value, and provide a way to construct a value with given units.

```javascript
function abs(x) {
  if (x >= 0*x) x else -x
}
function sqrt(x) {
  r = halfunit(x);
  while (abs(r*r-x) > x/1.0e9)
    r = r - (r*r-x) / (2*r);
  return r;
}
function halfunit(x) {
  u = unit(x); // [[meter,2]]
  i = 0;
  while (i < len(u)) {
    u[i][1] = u[i][1] / 2;
    i = i + 1;
  }
  // u = [[meter,1]]
  return one(u); // 1 meter
}
```

Units are checked for equality each time a value is stored back into a cell.

4. IMPLEMENTATION
We use PEG.js for parsing the programming language. Calling the parser with a string containing source code returns the abstract syntax tree (AST) that represents the code. The interpreter is written in JavaScript. Evaluating the AST finds the value of an expression. Expressions are evaluated using big-step semantics.

We use a jQuery-based spreadsheet for user interface. The current implementation can perform operations only on cells within the same row. The implementation makes an AST that represents the function. For computing the results, it makes an AST node that represents a function call and evaluates the AST node to perform the computation.

An implementation of Abacus is currently available on the web at URL http://assortedtrails.com/abacus/js/jquery/test.html

5. RELATED WORK
Forms/3 [1] is a complete programming framework using the ideas of spreadsheets and cells. It provides a sophisticated programming model with a lot of power. Users can create arbitrary objects using cells as the basic building blocks and then define operations on them. However, the steep learning curve for the system might be a deterrent in using it for practical applications.

Sestoft [5] introduces the idea of function sheets in spreadsheets. He provides a nice visual representation and talks about recursive and higher-order spreadsheets. However, the
function sheets in his framework are separately stored from the data.

Wakeling [6] uses ideas from functional programming to provide programmability in Microsoft Excel. It allows users to define Haskell functions, to use these functions in formulae, and to evaluate these formulae in spreadsheets.

Kennedy [3] presents the major ideas used for doing type checking with units.

6. FUTURE WORK

We are looking at several directions of extending the work.

First, there are many things that can be added to the visual interface. One is to add the ability to create independent function tables within the sheet that are not linked to the other cells. It would be good to have some mechanism to link functions together and create higher level abstractions. Another interesting idea would be to add the ability to create classes and objects.

Like traditional spreadsheets, Abacus should allow arbitrary arguments given to formulas, rather than all inputs and the outputs on the same row.

By restricting the allowable formulas in cells to a function call with arguments from neighboring cells, we could represent each formula visually instead of textually, allowing the user to visually confirm the input cells that affect an output cell. In traditional spreadsheets, the user must decipher a row/column code such as D13, making it harder to verify that formulas are correct.

Currently, each argument to a function is a single scalar value. Traditional spreadsheet programs allow arrays as arguments (e.g. sum(A1:A13)) so that an entire array of cells can be input to a function.

Spreadsheets typically offer a variety of formats for displaying numerical values, such as in percentages or dollar amounts. With each cell having a specified unit, we could restrict the available formats to those that make sense for the cell, such as allowing percentage formats only for dimensionless numbers or monetary formats only for monetary units.

It is currently not possible to specify a fractional unit power within the language or user interface, although it is possible to create a value with a fractional unit power at runtime. The sqrt function above will work with fractional units (e.g. sqrt(100 meter) is 10 meter^-0.5), but they may not work properly with fractional power that are thirds because they are implemented as floating point and may not compare as equal.

In the language and type system itself, we can add automatic conversion of units. So for example, when adding meters to feet, both values should be normalized to a single unit and the results should be reported in that unit. We can also have unit aliases, for example, Newton can be an alias for kg · m/s^2. We can integrate static type checking into the system in a limited way for fundamental types like Number, Boolean, string, array etc. We could supplement this with a scheme for declaring the type of each argument and use type interference to determine the type of each expression and the return type.

Lastly, error reporting is an area where we can add a lot of improvement to the system. The error messages can be made much more helpful by adding more information about the reasons for the error and pointing to the relevant section of the code.

7. REFERENCES


