Extend

Project Proposal

Jared Samet, Kevin Ye, Nigel Schuster, Ishaan Kolluri

jss2272, ky2294, ns3158, isk2108

Table of Contents

Inspiration

Tools such as Microsoft Excel, Google Spreadsheets, and Lotus 1-2-3 have throughout the past two decades served as a way for both individuals and teams to properly aggregate and compute on large amounts of data. However, we are in an age where these tools are being used to perform calculations on troves of data that only gets more complicated with time. After speaking with individuals well versed in spreadsheet technology, discussing their pain points, and reflecting on our own experiences, we found that a primary need was the ability to encapsulate a long sequence of calculations within the cells of a worksheet as reusable and applicable on multiple datasets. Discussing ways to solve this need ultimately gave birth to Extend.

While typical spreadsheet applications have restrictions that limit their use as general-purpose programming environments, they have three features in addition to their GUIs that make it easy for relatively unsophisticated programmers to perform complex computations and input manipulations:

- **declarative syntax** - The user doesn't have to specify calculation order; the software automatically figures this out on its own by examining the dependencies among the cells.
- **immutable values** - Typical formulas are side effect free, which makes it easier to understand the calculation logic.
- **automatic formula adjustments** - A user will often have certain inputs arranged in a tabular form. Once they have constructed a formula that produces the correct output for a single row, the spreadsheet software generally makes automatic adjustments to the formula that should produce the correct output specific to neighboring input rows.

At the same time, the inability to encapsulate complex calculations as functions, and compose and reuse these functions, imposes severe limitations on the uses of spreadsheets. In addition, a spreadsheet cannot be compiled and used as a standalone program; to perform a calculation, a user needs to actually open the application and manually enter the appropriate inputs into the appropriate locations, or use an entirely separate scripting language to effectively do the same thing.

Extend is our attempt to create a programming language that incorporates the easy development advantages of spreadsheets while allowing users to create standalone applications that allow for true function composability and reusability.

Description

Extend is a declarative programming language, where the developer can designate a set of calculations as a reusable function. Variables are fixed-size two-dimensional ranges of cells; each individual cell contains a formula, which may

be shared with neighboring cells. The dependencies between cells are used to calculate the order of calculation, which does not need to be specified by the developer. Extend allows cells to refer to other cells both by absolute and relative positions, as in spreadsheets, but borrows the slice syntax from Python to describe ranges in compact but flexible ways. Functions on ranges of cells will be applied hierarchically when there are dependencies. The arguments to a function consist of zero or more cell ranges, and the return value is a single range; all of these ranges can be of variable length. Functions can use other functions as part of their dependent calculations; each of these can create its own variables within its own local scope to perform calculations. The typing is simple, as the language has a very specific set of use cases. Extend compiles to LLVM.

Use Cases

Case 1: You have a set of complicated calculations associated with computing the discounted cash flow of a company. You'd like to apply the same set of calculations to another company, but you cannot easily do this. With Extend, you'd be able to declare a "discounted cash flow" function that takes a projection of cash flows associated with the company, and then apply this function repeatedly to as many other companies as desired. With a compiled Extend program, you can reuse this function as new information comes in.

Case 2: Within a company, there are domain-specific calculations that many people may have to do. Extend offers the ability for people to internally share programs that solve the company's common technical challenges, and improves productivity by making it possible to write libraries with the same development model that they are already familiar with from spreadsheets.

Case 3: Extend makes it significantly easier to schedule and automate computations on constantly changing input data. A human is required to use a spreadsheet to manually update inputs in order to calculate the updated results.

Therefore, calculations that need to be periodically performed on a dynamic set of inputs can be easily automated with a compiled executable in Extend.

Components & Syntax

Extend, being a focused language, leverages a small range of primitives. It bridges the gap between in-depth knowledge and simple syntax. Thus, it borrows elements from Java, Python and Excel.

Data Types

Extend has two classes of data: cells, which contain either a single number or a single formula that evaluates to a number, and ranges, which are a two-dimensional set of cells arranged in rows and columns. A variable always refers to a range. Since cells are immutable, all values are passed by reference. This further simplifies the use of the language and reduces errors when comparing values. Equality between two cell ranges is established by value. If a cell that is read does not hold a value, it returns "empty". "Empty" does not throw an error; instead, it propagates throughout further usage of the value. At compile time, the compiler will assert that the dimensions of the cell ranges passed into a function and returned from a function match the declared dimensions. Overall, the design greatly flattens the learning curve for a new user.

Cell Range: `foo[1, 2]` for a single cell or `foo[2:[4],3]`, where `[4]` is a relative reference and resolves to (referring cell's row)+4.
Number: `someValue = 2` or `someValue = 4.3`

Functions

Functions lie at Extend's core; however, they are not first class objects. Functions take the following form: a function starts by optionally declaring the dimensions of its return range. If no range is specified, then the return value is a 1x1 range consisting of a single cell. The function name and its arguments follow. The

declaration style of the return value is the same for function arguments. Since it can be verbose in writing certain operations, Extend will feature a comprehensive number of built-in and standard library functions. An important built-in functionality is I/O; the user can either do a raw read where each character in a given file occupies a cell in a column or specifies delimiters to read in a matrix. Writing to a file is handled in the same fashion. As Extend features a declarative programming style, functions for a specific cell will be evaluated once the value of the cell is read, not when the formula is inserted into the cell. The main entry point of any program is a main function of the form "`main([_, _] args)`", where each column represents one argument.

A function is composed of four parts. The first is a declaration, which specifies the dimensions of the return value and of any arguments. The next three, which are inside the body of the function, are: (1) a set of variable declarations, which specify the dimensions of any local variables; (2) a set of formula assignments, which determine the contents of individual cells; and (3) a return statement, which specifies the range to be returned by the function. The order of the statements inside the body of the function is arbitrary; the compiler will determine which cells will be evaluated and in what order.

A variable declaration of the form `[m, n] foo;` indicates that the variable foo is a range consisting of m rows and n columns.

Formula assignments take the form: `foo[row_slice, column_slice] = expression;` and indicate that `expression` applies to every cell in the subrange of foo specified by `[row_slice, column_slice]`.

As in Python and Go, a slice has the form `start:end` in which start is inclusive and end is exclusive; in other words, the length of a slice is (end - start). Both start and end are optional and default to zero and the dimension length, respectively. In addition, they can be negative, with the same interpretation as in Python. If a single integer `i` is specified, without a colon, it is equivalent to `i:i+1`.

Expressions allow for arithmetic and boolean operations, function calls, conditional branching, and extraction of contents of other variables.

Supported arithmetic and boolean operators include the common `+, -, *, /, %, !, &&, ||, ^^` and generally have the same interpretations as in C, C++ and Java. Conditional branching is provided by the `condition ? value-if-true : value-if-false` ternary operator, where the `: value-if-false` clause is optional and defaults to `empty` if absent. More complicated conditional branching is provided by the `switch` keyword, which works similarly to its use in Go; `switch` can either be used to compare the value of an expression to a list of cases, or can be used to choose the expression corresponding to the first condition that evaluates to a truthy value.

Variable content extraction is provided with the syntax `bar[row_slice, column_slice]`, but there is additional flexibility; either the `start` or the `end` of a slice can be specified either in absolute terms or relative to the index of the assignment cell. Relative indexing is specified by enclosing an index in square brackets. If a row slice or a column slice is entirely omitted in an extraction expression, it defaults to the relative value `[0]` if that dimension has length greater than 1, and the absolute value `0` for a dimension of length 1.

Examples:

```
singleCellReturn() {
  return 3;
}
/* This function is identical to having written:
[1,1] singleCellReturn() {
  [1,1] foo[0,0] = 3;
  return foo;
} */

[5,m] rangeReturn([1,m] input) {
  [5,m] foo[0,:] = input; // First output row equal to input
```

```
  foo[1:,:] = foo[[-1],] + 1; // Next four are input+1, input+2, etc.
  return foo;
}
/* Equivalently:
[5,m] rangeReturn([1,m] input) {
  [5,m] foo[0,0:m] = input;
  foo[1:5,0:m] = foo[[-1],[0]] + 1;
  return foo;
} */
```

It's important to reaffirm that Extend is a declarative language. The compiler will first look to the range specified by the `return` keyword to determine which calculations will be performed, and in what order. The body of a function can be thought of as a single expression, while different parts of the expression can be fragmented into simpler expressions.

Compilation

The compiler will build and topologically sort a dependency graph (rooted at the return value) between the variables and cells of an Extend function. The machine language instructions to calculate the values for each cell will be executed in the resulting sort order. If multiple branches of the graph are independent, they could be executed in parallel.

Example Usage

DNA Sequence Alignment using Dynamic Programming Algorithm

```
/* Computes the optimal alignment of two character sequences based
   on the specified reward for correct matches and penalties for
   mismatches / gaps in alignment */
[_,2] computeSequenceAlignment(
      [m,1] seq1, [n,1] seq2,
      matchReward, mismatchPenalty, gapPenalty) {

      [1,n] seq2T = transpose(seq2); // part of stdlib
```

```
[m+1,n+1] score;
[m, n] scoreFromMatch, scoreFromLeft, scoreFromTop;
[m, n] step, path;

score[0, 0] = 0;
score[1:,0] = score[[-1],] + gapPenalty;
score[0,1:] = score[,[-1]] + gapPenalty;
score[1:,1:] = nmax(scoreFromMatch[[-1],[-1]],
     nmax(scoreFromLeft[[-1],[-1]], scoreFromTop[[-1],[-1]]));

scoreFromMatch = score + (seq1 == seq2T) ?
     matchReward : mismatchPenalty;
scoreFromLeft = score[[1],] + gapPenalty;
scoreFromTop = score[,[1]] + gapPenalty;

step = (scoreFromMatch >= scoreFromLeft) ?
     ((scoreFromMatch >= scoreFromTop) ? 'D' : 'T') :
      (scoreFromLeft  >= scoreFromTop) : 'L' : 'T'));

path[-1,-1] = 1;
path[-1,:-1] = (step[,[1]] == 'L' && !isEmpty(path[,[1]]) ?
     1 + path[,[1]] : empty;
path[:-1,-1] = (step[[1],] == 'T' && !isEmpty(path[[1],]) ?
     1 + path[[1],] : empty;
path[:-1,:-1] = switch { // Golang style
     case step[[1],[1]] == 'D' && !isEmpty(path[[1],[1]]):
          1 + path[[1],[1]];
     case step[,[1]] == 'L' && !isEmpty(path[,[1]]):
          1 + path[,[1]];
     case step[[1],] == 'T' && !isEmpty(path[[1],]):
          1 + path[[1],];
     };

pathLen = path[0,0];
[m, 1] seq1Positions = pathLen - rmax(path[,:]);
[1, n] seq2PositionsT = pathLen - rmax(path[:,]);
[n, 1] seq2Positions = transpose(seq2PositionsT);
[pathLen, 1] resIdx = colRange(0, pathLength); // stdlib
[pathLen, 1] seq1Loc = match(resIdx, seq1Positions); // stdlib
[pathLen, 1] seq2Loc = match(resIdx, seq2Positions); // stdlib

[pathLength, 2] results;
```

```
      results[:,0] = seq1[seq1Loc];
      results[:,1] = seq2[seq2Loc];

      return results;
}

/* Another example - part of standard library. Returns a column
   range with the numbers start...stop-1 */
[_, 1] colRange(start, stop) {
      [stop-start,1] res;
      res[0] = start;
      res[1:] = res[[-1]] + 1;
      return res;
}

/* GCD */
gcd(m, n) {
      return (m % n == 0) ? n : gcd(n, m % n);
}

/* Binary Search
   Hard to test this one without a working compiler!
   Because everything is passed by reference, the
   slicing should take O(1), not O(n). */

binSearch([m,1] list, val) {
      mid = floor((m-1)/2);
      return switch {
      case list[mid] == val:
            mid;
      case list[mid] > val:
            mid > 0 ? binSearch(list[:mid], val); // otherwise empty
      case list[mid] < val:
            m > 1 ? mid + 1 + binSearch(list[mid+1:], val);
      }
}

/* Matrix multiplication */

dot([m,1] v1, [m,1] v2) {
      [m,1] products = v1 * v2;
      return sum(products); // stdlib
}
```

```
[m, p] mmult([m, n] m1, [n, p] m2) {
     return dot(transpose(m1[,:]),m2[:,]);
}
```