Spreadsheet formula calculation – algorithm explanation

(Warning: code here is pseudocode, may contain some errors, but I believe it to be conceptually correct.)

Basics:

A formula in a spreadsheet looks something like this:

= 1 + 3

This is just about the simplest formula we could have, but the way it breaks down is exactly the same way that more complicated formulas break down (once we chop off the leading ‘=’ – that part just tells the spreadsheet that it’s looking at a formula rather than a string or value literal). Every expression follows the basic pattern:

Expression = Expression Operator Expression

For the sake of this explanation, we’ll define an expression as follows:

Expression: A piece of a formula that can be evaluated

Obviously the ‘1’ and the ‘3’ in ‘1 + 3’ fit this definition, but critically, so does ‘1 + 3’ as a whole. An expression therefore has a recursive definition that allows for an expression to represent much more complex constructions:

**1 – 3^(5/3)**

We break this down starting at the deepest level – the piece that would be evaluated first with standard order of operations.

In this case:

expression A = 5 / 3

expression B = 3 ^ A

expression C = 1 – B

Now for creating an engine that handles spreadsheet calculations, we want to be able to execute this quickly. This means doing this in a form that has little to no flow control, and which features pieces that a JIT compiler can easily optimize.

The first step is to have a parser convert the formulas themselves into json along the following lines:

Expression = {

type: ‘binary’,

A: {

type: ‘value’,

value: 1;

},

operator: “sub”

B: {

type: ‘binary’,

A: {

type: ‘value’,

value: 3;

},

operator: “exp”

B: {

type: ‘binary’,

A: {

type: ‘value’,

value: 5;

},

operator: “div”

B: {

type: ‘value’,

value: 3;

}

}

}

}

Yes, this is verbose. It’s also not trivial to construct for every formula in a large spreadsheet, many of which will be nested many levels deeper than this. Deciding whether the construction of this happens in the client at runtime or whether it’s preprocessed on the server (or whether we want the flexibility to support both)cons is a conversation for later.

Then part is constructing a method for turning that json into a calculation:

Expression.getEvalFunction = function(expression) {

switch(expression.type) {

case ‘binary’:

var constructFunc = Expression.operatorConstructorFunctions[‘op’ + expression.operator];

return constructFunc(expression.A, expression.B);

break;

case ‘value’:

if(Expression.valueFunctions[expression.value]) {

return Expression.valueFunctions[expression.value];

} else {

var type = Expression.getTypeFromValueString(expression.value);

var constructFunc = Expression.valueConstructorFunctions[type];

return constructFunc(expression.value);

}

break;

//handle other cases (unary operators, lists)

}

}

Expression.valueFunctions.number = function(value) {

var val = parseFloat(value);

return function() {

return val;

}

}

Expression.operatorConstructorFunctions.add = function(A, B) {

//here’s the recursion – make sure we have

var funcA = Expression.getEvalFunction(A);

var funcB = Expression.getEvalFunction(B);

return function() {

return funcA() + funcB();

}

}

// Same for other basic operators

This is a large code chunk, so I’m going to give an overview of what’s happening, and then go through the two specific cases:

1. Starting depth first (due to the recursion) we’re constructing functions that evaluate the result of each expression in the hierarchy.

2. Once the child expression functions have been built, the parent function gets built using the expression functions from the children as building blocks

3. Repeat until we have a single function that evaluates the whole thing.

Building the functions for values is straightforward enough (… for now.)

Looking instead at the process for constructing a binary function, here’s what we have:

case ‘binary’:

var constructFunc = Expression.operatorConstructorFunctions[‘op’ + expression.operator];

return constructFunc(expression.A, expression.B);

break;

and then later, the definition of the constructor function (the example is addition, but the differences between them are trivial):

Expression.operatorConstructorFunctions.add = function(A, B) {

//here’s the recursion

var funcA = Expression.getEvalFunction(A);

var funcB = Expression.getEvalFunction(B);

return function() {

return funcA() + funcB();

}

}

We look up the appropriate constructor function for the given operator, and then allow it to create the evaluation function (keeping the eval functions for its children available as closures);

The process isn’t actually \*that\* complex – the most confusing piece is that we’re keeping a lot of the pieces used to evaluate the functions ‘hidden’ as they’re closures within closures within closures. On the plus side, this results in a relatively clean top level where there’s just a single evaluation function for each expression.

Now, so far, so good. This piece is actually quite easy. There are some edge cases that need to be dealt with in order to properly implement just the mathematics (unary operators, for instance). The piece that adds complexity comes when we start adding the spreadsheet specific functionality to the model.

The most fundamental thing that needs to be added in order to bring spreadsheet functionality are external cell references. These are references either to specific cells in the spreadsheet, or to a range of cells in the spreadsheet. They can be literal references of the form ‘[worksheetName!]CellAddress’ or named ranges (essentially a variable name that has a literal reference as its definition). When the address represents multiple cells, it’s in the form: ‘[worksheetName!]TopLeftCornerCellAddress:BottomRightCornerCellAddress’

This might cause our example formula from earlier to look like

**1 – 3^(capTotalUnitsSold/E5)**

capTotalUnitsSold is a rangename reference, and E5 is a direct cell reference to a cell on the current page. (you might also see a direct cell reference written as $E5 E$5 or $E$5 – for the purposes of calculation, these are irrelevant, but when working in Excel that stops the reference from changing when you copy the formula to a new location)

When worksheetName is not included, the reference is assumed to be on the current sheet. Which, of course, means that when we’re evaluating an expression we need some context for it (what location is it being evaluated from? For more reasons than just this one, knowing this is important). This means we’re going to need to make some metadata available to our evaluation.

Of course, this alone adds a fair bit of complexity. For one, we need to keep track of the locations of each expression. What is more, we need to be able to pull data that is offset from a given location (and calculate that efficiently, as offsets can vary at calculation time). Lastly, our evaluation process needs to be modified so that the current calculation context is available to any constructor functions that need it.

One example of this:

Let’s say we have a rangename ‘calcTotal’ defined as ‘sheet1!E5:I5’ which is a range of 5 cells

and then we have an expression ‘=5 + calcTotal’

So… which calcTotal? How are we adding a scalar to what appears to be an array of 5 different cells?

This is where the context becomes important. ‘calcTotal’ indeed is an array of 5 different cells, but when it’s used in a context that wants a scalar, it can be used to reference just one of those cells *depending on the location that it is being evaluated from*. Specifically, if the cell we’re evaluating from shares either a column or a row with ‘calcTotal’ – and if only one of the values in calcTotal is in that column or row – it will use that value. (It can’t share both – if it shared both a column and a row, then the cell we’re calculating from is inside calcTotal and that would be a circular reference. Those aren’t allowed)

So we need to make sure that whatever our evaluation function for calcTotal is, that it allows us to (efficiently) get different things from it depending on context. If we call it in a context where we need a scalar, it needs to return the scalar appropriate to that context. We also will likely need to be able to construct these functions efficiently on the fly, as there are functions (a piece that hasn’t been covered yet) that output references rather than values (confusingly, many of these functions \*appear\* to output values in normal use)

This probably means introducing a context parameter that gets passed throughout the calculation hierarchy. This need to include at minimum:

location: the location being calculated from, worksheet and cell address (this is the same for every operation in a hierarchy)

requiredType: \*IF\* there’s a type the parent operation needs (for instance the ‘+’ operator requires scalars) we need to pass it along.

Right now, I’m leaning towards building things in such a way that the context is available within the closures that need them. This is one of the least finalized pieces conceptually, though. The flash version of this calculator did its context evaluations at build time, and I’d prefer to move this to the point The last big thing we need to deal with – and one of the big features of spreadsheets – are worksheet functions.

Going back to our example formula:

**1 – 3^(capTotalUnitsSold/SUM(E5, E7, E9, E11:E15))**

Functions are expressions that accept a comma delimited list of arguments (each of which is also an expression)

This brings our list of what an expression can be to the following:

1. A Value
2. A reference to a location in the spreadsheet
3. An expression, an operator, and a second expression
4. A function followed by parentheses containing a list of expressions separated by commas
5. Parentheses enclosing an expression (Actually, this probably doesn’t exist post-parsing, as any parens that are solely for specifying order of operations should be redundant)

Our json for functions likely looks something like this:

{

type: ‘function’,

functionName: ‘SUM’,

arguments: [{ type: ‘reference’,

worksheet: ‘cap’,

row: 5,

column: 15,

endRow: 9,

endColumn: 15

}, { type: ‘reference’,

worksheet: ‘cap’,

row: 12,

column: 15,

endRow: 9,

endColumn: 15

}]

}

The constructor for functions will therefore look something like the following:

Expression.constructFunction = function(funcName, argumentArray, context) {

var context = context;

var func = Expression.worksheetFunctions[funcName];

var paramArray = [];

for(var i = 0; I < argumentArray.length; i++) {

paramArray.push(Expression.getEvalFunction(argumentArray[i], context));

}

}

Expression.constructFunctionWrapper = function(func, args, context) {

var worksheetFunction = func;

var functionArgs = args;

var expressionContext = context;

return function() {

//no real way to include the context only as a closure here.

worksheetFunction(args, expressionContext);

}

}

And an example function:

//all cap naming convention is from the spreadsheet – keeping it intact so that it’s clear that the function is a direct implementation of a worksheet function

Expression.worksheetFunctions.SUM = function(args, expressionContext) {

var sum = 0;

for(var I = 0; I < args.length; i++) {

args[i] ().each(function() {

sum += this.toNumber();

});

}

return new jsCalcNumValue(sum);

}

This looks slightly different than the simplified operations we were creating at the beginning – this is because it’s incorporating the concept of evaluating context dependent values at calcTime. Additionally, arguments to the SUM function can be either single numbers or ranges of numbers. In order to deal with this, I’m introducing a more complex type for handing values back from expressions – jsCalcValue. jsCalcValue’s intent is to allow us to deal cleanly with different data types (and by cleanly, I mostly mean avoiding redundant type checking code in every function implementation). It provides facilities for type conversion where appropriate (If a string value is in the midst of the data we’re summing, toNumber will return a 0 to the sum (this is what Excel does)) and also for masking the difference between values and references.