Towards Wide-Spectrum Spreadsheet Computing

Enzo Alda
Lakebolt Research
Connecticut, U.S.A.
enzo@lakebolt.com

Javier López Lombano
Universidad Simón Bolívar
Zaragoza, Spain
javier@lakebolt.com

Mónica Figuera
The University of Bonn
Bonn, Germany
s6mofigu@uni-bonn.de

Juan Andrés Escalante
Universidad Simón Bolívar
Madrid, Spain
jaescalante02@gmail.com

Richard Lares Mejías
ULACIT
San José, Costa Rica
richardlm57@gmail.com

Pablo Maldonado
Universidad Simón Bolívar
Toulouse, France
prmm95@gmail.com

Jacquin Mininger
Lakebolt Research
New York, U.S.A.
jacquin.mininger@gmail.com

Jaques Frenkel
Lakebolt Research
New York, U.S.A.
yshfrenkel@gmail.com

***Abstract*—with the recent additions of dynamic arrays and user-defined functions, Microsoft Research has taken two important steps towards the unification of spreadsheets and general purpose programming environments. We offer our perspective on that quest, reiterating that the full potential of the spreadsheet computing paradigm can only be realized after removing the entanglement of the computational model and its visualization. Our experimental findings suggest this can be done without detracting from the live and user-friendly nature of the spreadsheet programming experience.**

Keywords — end-user development, wide-spectrum computing, spreadsheets, live programming

I. Motivation

Spreadsheets are the most popular end-user computing environment. When Visicalc first appeared in May 1979, it quickly became a killer app: spreadsheets are widely credited as a major catalyst of the personal computing revolution.

Academic research [1] and our own personal experience suggest that most spreadsheet practitioners don't think of themselves as programmers. Our own anecdotal evidence is that MS-Excel practitioners who self-identify as programmers are the few who have successfully crossed the divide between the world of worksheet formulae and Visual Basic (VBA). Nevertheless, the language of MS-Excel formulae by itself is a functional programming language [2] [3] albeit one that did not support functional abstraction until very recently.

The logical conclusion is that MS-Excel is the most popular programming environment in the world [4]. In principle, that alone should make spreadsheets primary candidates for teaching programming in schools and creating computing environments well-suited for a wide range of users. In practice, spreadsheets are rarely considered seriously for such purposes because of the shortcomings discussed here.

Our goal is to transform spreadsheets into general purpose reactive computing environments, thereby realizing their full potential. To that endeavor we extend the core functionality of spreadsheets with modern programming language concepts, honoring widely accepted principles of language design.

II. The Spreadsheet Core

The core of a spreadsheet document, i.e. a workbook, is a set of named worksheets. A worksheet consists of a grid of nameless cells referenced using A1 notation. Each cell contains an editable formula and displays the result of evaluating such formula consistently with the state of the entire workbook. We do not distinguish between cells containing values and cells containing formulae: values are fully reduced expressions. We use the terms "formula" and "expression" interchangeably, referring to the expression in a cell as the CVALUE of the cell, and the result of evaluating such expression in the current state of the workbook as its RVALUE. For completeness, empty cells contain a value known as *null*, represented with a question mark and visually rendered as an empty cell.

Our analysis of the core functionality of spreadsheets starts with the state-of-the-art before Microsoft made dynamic arrays generally available [5] and their announcement of lambda expressions and user-defined functions [6][7]. Doing the analysis this way simplifies the presentation and establishes a common baseline inclusive of products that were all too happy to imitate the MS-Excel core and are now left to catch-up.

A. The Language of the Spreadsheet Core

Any machine, virtual or tangible, that offers an interface for humans or other machines to operate on it, implicitly defines a language. Listing 1 shows the grammar of formulae in the core.

XLS.1) E → ? | <error> | true | false | <number> | <string>

XLS.2) E → <symbol>(E, ..., E)

XLS.3) E → <A1> | <symbol> ! <A1>

XLS.4) E → <A1>:<A1> | <symbol> ! <A1>:<A1>

Listing 1: abstract syntax of spreadsheet core formulae

Grammar symbols enclosed in brackets denote pseudo-terminals, i.e. elements that are terminal for syntax description purposes but represent a (possibly infinite) set of values, e.g. numbers. <A1> stands for worksheet coordinates given by the eponymous A1 notation (e.g. UX42, $C7, or A1); <symbol> denotes a function name in XLS.2 and a worksheet name in

XLS.3 and XLS.4. Following the time-honored tradition of excluding "syntactic sugar" from an abstract syntax, XLS.2 stands for all possible function invocations, including the ones using infix operator notation.

We can describe a set of worksheets listing each non-empty cell next to its formula. Doing so, we obtain a set of equations that completely specifies the state of the workbook. As stated by Simon Peyton Jones [2], such textual form reveals a pure functional program. The assertion is correct, but made from a static, stable state, perspective: i.e. ignoring model mutation. For the unification purpose stated in the abstract, we do take model mutation into account. Model mutation is usually performed via user interaction and we consider such interaction a key ingredient of the spreadsheet computing paradigm.

Specifically, we regard the act of changing a cell as an assignment. It could be argued that changing a cell should be considered a model (i.e. program) editing operation, and not a runtime assignment. Both positions are valid, given that the lines of the traditional edit-compile-run cycle are blurred in a live programming environment. But the hint that we should regard cell editing as the manual equivalent of a runtime assignment is given precisely by the existence of imperative adjuncts, like VBA and AppScript. Such extensions were hastily "plopped" on top of the core in response to fast growing market demand; with little, if any, regard for elegance and consistency. Horrible as we feel they are, the imperative programming adjuncts are useful and are prima-facie evidence of the need to automate computational effects that otherwise can only be accomplished via user interaction.

The arbitrary nature of the imperative extensions was one of the factors motivating ZenSheet [8][9], the experimental spreadsheet project tied to our research. We stipulate that, as a *minimum requirement*, the language of expressions allowed on the right hand side of an assignment must be the same as the language of formulae allowed in spreadsheet cells. That's not the case for imperative adjuncts in commercial spreadsheets. Listing 2 below describes the possible actions in the core.

XLS.5) A → **add** <symbol>; | **remove** <symbol>;

XLS.6) A → <symbol>!<A1> := 'E';

XLS.7) A → <range> := values(<range>);

XLS.8) A → <range> := formulae(<range>);

XLS.9) A → <alter>(<symbol>, dim, start, n);

where <range> → <symbol> ! <A1>:<A1>

and <alter> → **insert** | **delete**

Listing 2: abstract syntax of actions in the spreadsheet core

XLS.5 denotes the addition and removal of worksheets. XLS.6 is the single cell formula assignment: the expression on the right hand side is quoted to indicate that the *expression*, not the *result* of evaluating it, becomes the CVALUE of the cell on the left hand side. XLS.7 and XLS.8 represent the range copy-and-paste of values and formulae respectively. XLS.9 represents insertion and deletion of rows and columns in a worksheet: *dim* identifies the dimension (row or column), *start* is the starting position for the operation, and *n* is the number of rows or columns added or removed. Technically, *dim*, *start*, and *n* are all integers and should simply be <number> in the abstract grammar. We just replaced the pseudo-token with a descriptive name, to make the grammar more expressive.

There are other ways to alter the state of a workbook. For brevity, we won't consider mechanisms that bind cells to real-time data feeds. Also, stochastic functions, e.g. random number generators, and time-dependent functions, conceptually change the state of a model continuously. In practice they can do so only at certain points in time. A discretization of the compute cycle is helpful not only for implementation purposes, but also for verifying consistency. Cells dependent on non-memoizable functions are marked as inconsistent at the beginning of each compute cycle, letting the propagation algorithm do the rest.

That's the spreadsheet core. It is fairly compact and easy to understand. It fulfills the desirable design goal of conceptual simplicity but is inadequate for serious programming, as we will show. It is worth noting what's *not* part of the core. First, no imperative adjuncts; second no plugins/add-ins like the Solver and Power Query. Such additions are valuable for users, but we argue that, at least in principle, it should be possible to implement them with the same tools provided to end users. Hence, we don't consider them part of the core.

There is an operation we have not included in the core that deserves debate: the range cut-and-paste. It is tempting to think that a cut-and-paste operation from range *A* to range *B* can be modeled with a copy-and-paste from *A* to *B* followed by assigning null to all the cells in *A* - *B*, but that is not correct. If a cell *c* contains a formula that refers to one or more cells in *A*, those references will be adjusted to "follow the cells" being moved. That's not the case with copy-and-paste, where no such adjustments are necessary, nor wanted. Turns out cut-and-paste is very different from copy-and-paste when we account for A1 behavior, and is also a pretty common operation. Then, why is cut-and-paste excluded from the core? It is excluded because, when used properly, cut-and-paste is just a layout management operation: a visual rearrangement of what is being computed. It is necessary in traditional spreadsheets only because of the entanglement of the model and the view, as we'll see later.

B. Critique of the Spreadsheet Core

As originally stated by us in 2017 [8][9] the spreadsheet core suffers from three fundamental issues.

1) Lack of Functional Abstraction

A simple inspection of listings 1 and 2 reveals that all function calls (XLS.2) identify the function with a symbol. These symbols must correspond to predefined functions, since the only objects users can create and name are worksheets (XLS.5). Hence, all functions are predefined: "Spreadsheets lack the most fundamental mechanism that we use to control complexity: the ability to define re-usable abstractions", as Simon Peyton Jones et al. nicely stated in 2003 [2].

Even more nicely, that's no longer the case for MS-Excel: on December 2020, Microsoft announced [6][7] the addition of lambda expressions and the ability to bind symbols to them,

effectively adding user-defined functions to the core. This is a very important milestone in spreadsheet evolution: to the best of our knowledge, MS-Excel is the first, and so far the only, *commercial* spreadsheet capable of functional abstraction.

2) Overly Simplistic Type System

Traditional spreadsheets are unitype, or dynamically typed as many prefer to say: there is no way to constrain the type of worksheet cells. From a static perspective, there is one and only one type we can guarantee each cell to have, and that type is "anything goes": all cells are unlabeled variants.

Per listing 2 above, worksheets are the only entities that can be named by users (XLS.5) as well as the only data structure available to them. All conceptual variables in a computation model – e.g. an interest rate, a mortgage record, or a vector of payments – must be mapped to a cell region and referenced via coordinates, using A1 notation. This programming model leads to what we call the "coordinate soup problem": a mess of hard to read formulae, which often makes spreadsheet maintenance a painful exercise in reverse engineering.

The coordinate soup problem above is only partially alleviated by using name aliases for cells and ranges. For brevity we won't go into specific issues working with aliases: suffice to say that they are a poor replacement for true model objects with a proper identity.

A1 notation should be considered harmful and its use eradicated. Nevertheless, ZenSheet has supported A1 notation since 2017: we find it valuable to help spreadsheet practitioners gradually learn the richer modeling experience and model-view separation ZenSheet provides. It makes possible to demonstrate refactoring transformations that make spreadsheet models easier to read and resilient to layout changes; e.g. separating conceptual model variables from worksheets.

In 2019, Microsoft Research announced the availability of dynamic arrays [5], thereby starting the deprecation of the now legacy CTRL+SHIFT+ENTER (CSE) array formulae. It was the first significant evolution of the spreadsheet core in decades, and a much welcome one. Along with dynamic arrays, Microsoft Research also announced their future plan to support "vectors, records, and even domain-specific data types implemented by third parties." It is still unclear to us how end-users will be able to define their own types.

We have reservations about the way Microsoft is making progress towards a richer type system in spreadsheets. Our reservations stem from the apparent reluctance to address the entanglement issue, which we are covering next. However, the announcements show that Microsoft is actively committed to enrich the type system of the MS-Excel core, and that alone is reason for celebration.

3) Entanglement of the Model and the View

As spreadsheet practitioners graduate to more advanced levels of proficiency, devising best practices to make their conceptual models more transparent and easy to maintain, they encounter a fundamental issue: worksheets play a double role in spreadsheet computing. As we have shown, they are an integral part of the computational model, indeed the top level named elements of the concrete model, but worksheets are also the means for model visualization and editing: they play a role as grid layout managers with basic IDE functions.

Spreadsheet errors are sometimes introduced by efforts to improve the layout using cut-and-paste operations. They can also be introduced by attempts to manage shared variables from multiple worksheets. A model variable, be it a single cell or a range, can truly exist in one and only one worksheet and the entanglement implies that it can only be edited there. For instance, assume a variable φ is a shared parameter of a model with multiple scenarios. The model computes a blended measure μ based on N scenarios, where each scenario is laid on a separate worksheet for modularity purposes. To help keep consistency across scenarios, we may want to designate a separate worksheet for the parameters, labeling its panel tab "Parameters". Moreover, we would like to visualize some shared parameters, like φ, no matter which scenario we are currently working on: it makes visual validation easier, sparing us from having to switch to the Parameters panel frequently. It is also good for presentation purposes: if we print a scenario we want φ to be visible. This can be easily accomplished by designating one cell in each scenario as a local copy of φ that references the shared φ; unfortunately, this can turn into a trap. For expediency, and also because scenarios often evolve from a worksheet that existed before parameters were factored out, it is tempting to refer to the local copy of φ rather than the shared φ in our "single source of truth": the Parameters worksheet. The trap is set: if someone changes the copy of φ in a scenario, to perform a "what if" analysis or a quick test, and then forgets to reset it, failing to realize that putting a number there clobbered the reference to the shared φ, we end up with an inconsistent model: one scenario is now using a different value for φ. This will affect the blended measure μ but likely not by much: only one scenario is using one inconsistent parameter. Such errors can easily go undetected for years.

The model-view entanglement is the root cause of various problems that affect end-users as well as researchers advancing the state-of-the-art in spreadsheets. Here is a quote from Microsoft's own preview of dynamic arrays in MS-Excel [10]:

"Don't worry about the spill range overlapping your data--if there isn't enough space, the formula will roll up and show an informative #SPILL error. When you select the #SPILL error, the formulas desired spill range will be indicated by a dashed blue border. Just move or delete the obstructing data and your formula will automatically spill."

The chosen remedy will likely be a *move* – a *cut-and-paste* operation: another example of asking end-users to: 1) act as memory allocators and re-allocators – because instead of modeling their conceptual variables directly they must map them onto cells of worksheet memory; and 2) act as layout managers – because forced to use worksheets as their visual grid layout. The entanglement has persisted for 42 years.

An advanced spreadsheet computing environment with proper model-view separation, like ZenSheet, can never have spill errors. The visual rendering of the model is a function of the model, but the model itself does not depend on the view. Rendering generates visual areas that cannot interfere with the model, because the model is fully agnostic about the layout.

III. Generalization of the Spreadsheet Core

A. Abstract Syntax of a Generalized Spreadsheet Core

1) Types

ZT.1) T → **null** | **error** | **bool** | **number** | **string**

ZT.2) T → **fun**(T, …, T) => T

ZT.3) T → **array**[, …,] => T

ZT.4) T → **struct**(T, …, T)

ZT.5) T → **lazy** T

ZT.6) T → **var**

ZT.7) T → <symbol>

2) Expressions

XLS.1) E → ? | <error> | **true** | **false** | <number> | <string>

XLS.3) E → < A1 > | <symbol> ! < A1 >

ZSE.1) E → <symbol>

ZSE.2) E → λ(T <symbol>, …, T <symbol>) -> E

ZSE.3) E → E(E, …, E)

ZSE.4) E → (E, …, E)

ZSE.5) E → [E, …, E]

ZSE.6) E → E[E, …, E]

ZSE.7) E → E:E

ZSE.8) E → E..E

ZSE.9) E → 'E'

3) Actions

ZSA.1) A → **type** <symbol> = T;

ZSA.2) A → T <symbol> := E;

ZSA.3) A → E := E;

Listing 3: abstract syntax of our proposed generalization

The listing above describes the abstract syntax of Lilly: the language underlying our generalization of the spreadsheet core. A fair coverage of Lilly would be impossible in this paper. Here we just mention a few points that are particularly relevant to the generalization of the core.

A variable is given a type upon definition (ZSA.2), but its type can be variant (ZT.6) which allows it to hold any type supported by the language: a form of gradual typing [11]. As a side note, turns out *var* and *lazy var* are the same type, but proving that is beyond the scope of this paper.

Even though A1 notation is supported, there is no special worksheet type. There used to be one, as late as 2015, and we eliminated it. A worksheet is just a two-dimensional array of lazy variants. The one instance where we deliberately opted to break the orthogonality principle was the decision to support A1 notation *only* for two-dimensional arrays of lazy variants.

Lilly supports user-defined types (ZSA.1) albeit only with structural equivalence, making them type aliases rather than separate types on their own. Work on user-defined types with nominal equivalence is under way. Here is a type definition example that reiterates a key result of our unification approach:

type Worksheet = **array**[,] => **lazy var**;

Last, Lilly is still evolving: we believe a type system is not complete for engineering purposes until it has support for sum types, recursive types, and parametric type polymorphism.

B. Generalization correspondence

Table 1 below shows how expressions and actions of the spreadsheet core correspond to those of the generalized one. This is not proof of semantic equivalence, of course, but it does show, if only from an abstract syntax perspective, how everything in the spreadsheet core is covered, providing further insight about our generalization approach.

TABLE 1: SPREADSHEET CORE GENERALIZATION

Core Production	***Generalized Production***
XLS.1	XLS.1 (included)
XLS.2	ZSE.3
XLS.3	XLS.3 (included)
XLS.4	ZSE.7
XLS.5 **add**	ZSA.2
XLS.5 **remove**	ZSE.3 via ***sys.undefine*** function
XLS.6	ZSA.3
XLS.7	ZSA.3
XLS.8	ZSA.3 + ZSE.3 via ***formula*** function
XLS.9	ZSE.3 via ***insert*** and ***delete*** functions

sys.undefine is a system function in the ZenSheet VM that removes a symbol passed as argument; ***formula*** is a special VM function that returns the CVALUE of a variable; ***insert*** and ***delete*** are predefined functions in the ZenSheet library.

IV. Sample Models

All the models here described are available for review and experimentation via the ZenSheet Portal (portal.zensheet.com). After loading a model, use the *Cycle* button to make it come alive (automatically reevaluating stochastic variables), and the *Tick* button to manually advance one compute cycle at a time.

(Please contact one of the first two authors for technical issues)

A. Random Matrix Multiplication (rmm)

The *rmm* model multiplies matrices of changing numbers, with the added twist that the matrices also change in size. To achieve such purpose, three random integers (*M*, *N*, and *K*) are generated in each computing cycle within bounds given by a formula. The *lhs* matrix (on the left of the multiplication) has dimensions *M*x*N* and *rhs* (on the right) has dimensions *N*x*K*. A

user-defined matrix-constructor function, named *Matrix*, is called to generate each input matrix. Figure 1 shows the input matrices, with one of the slides displaying the *rhs*' formula. Figure 2 shows the resulting matrix, which is computed with the simple formula *mmult(lhs, rhs)*. Note that MMULT is also the name of the matrix multiplication function in MS-Excel.

The *rmm* model was originally conceived to demonstrate the limitations of traditional MS-Excel array formulae, entered via CTRL+SHIFT+ENTER (CSE), which have been in existence since the 90's. CSE array formulae require users to select a worksheet range in advance to hold the result of the array formula: i.e. the user must perform worksheet *memory* allocation for the result manually.

With Microsoft's introduction of dynamic array formulae, CSE formulae are now a deprecated feature, supported for backward compatibility. However, only some array functions support dynamic behavior already: Microsoft is implementing dynamic variants of the existing array functions.

As of this writing, the MS-Excel core still can't replicate the *rmm* model in a natural way: the function RANDARRAY supports dynamic behavior, and is well suited for generating the input matrices, but the function MMULT has not achieved dynamic status yet. By contrast, any ZenSheet function that can operate on fixed-size arrays can operate on dynamic ones, even multi-dimensional arrays that can change size along multiple dimensions. This is true for predefined as well as user-defined functions: by virtue of having disentangled the model from the view there is never a need for a separate implementation.

The *rmm* model was first demonstrated at the LIVE 2017 workshop (SPLASH 2017), and later at Lambda Days 2020. Video available [12]: *rmm* starts at 29:34

B. Three Triangles (3t)

This model demonstrates the computational as well as the user experience implications of supporting laziness in mutable containers. One panel in this model shows three triangular arrays of numbers, named t1, t2, and t4, built with formulae that depend on two random variables: x, y. Although the three triangles have different type, the results of evaluating them are identical by construction, even as they change in reaction to mutations of x and y.

TABLE 2: POSSIBLE LAZINESS COMBINATIONS IN AN ARRAY OF ARRAYS

Symbol	**variable type**
t0	array[] => array[] => double
t1	array[] => array[] => **lazy** double
t2	array[] => **lazy** array[] => double
t3	array[] => **lazy** array[] => **lazy** double
t4	**lazy** array[] => array[] => double
t5	**lazy** array[] => array[] => **lazy** double
t6	**lazy** array[] => **lazy** array[] => double
t7	**lazy** array[] => **lazy** array[] => **lazy** double

The model actually has 8 variables as shown in Table 2. One variable (t0) is not lazy and four (t3, t5, t6, t7) have nested laziness. Nested laziness appears to show promise for symbolic processing and generative programming: a development we did not anticipate when we started the ZenSheet project. The implications and obstacles involved are still unclear to us at this time: Lilly is an experimental language and is forcing us to explore possibilities that are sometimes mind-bending.

The *3t* model (albeit only with triangles t1, t2 and t4) was demonstrated at the LIVE 2017 workshop (SPLASH 2017); the snapshot in figure 3 highlights one sub-formula to illustrate the construction correspondence of the three variables.

C. Targeted Marketing Simulation (tmx)

The *tmx* model (figure 4) combines arrays, structures, user-defined functions and types; it was first demonstrated at Lambda Days 2020. Video available [12]: *tmx* starts at 37:11.

D. Higher Order Functions (hof)

hof demonstrates a functional map-filter-reduce pipeline, shown at the LIVE 2017 workshop and Lambda Days 2020. Video available [12]: *hof* starts at 27:50

E. Worksheet (wsheet)

The *wsheet* model (figure 5) demonstrates A1 notation, in-and-out of worksheet formulae, and $-correct copy & paste. Lambda Days 2020 video available [12]: *wsheet* starts at 15:40.

V. CLOSING REMARKS

We have presented evidence supporting our key diagnosis: three issues, which can be traced back to Visicalc – the very first implementation of the spreadsheet computing paradigm – have prevented spreadsheets from realizing their full potential. We have stated our position – namely that it is possible to overcome those issues by identifying the implicit language underlying the core spreadsheet functionality and generalizing it with modern programming language concepts. By "modern", we mean concepts that were discovered in the late 70s and 80s, finding their home in languages like ML and Miranda. We also stated that it is possible to do all this while honoring the live reactive nature of spreadsheets. For this purpose, we define a language, named Lilly, where variables of a lazy type can hold expressions that are not fully reduced, introducing the semantic distinction between CVALUE and RVALUE. Lilly can express traditional spreadsheet models that simply rely on the spreadsheet core. Last, we complement the Lilly VM with an AST-driven IDE, suitable for the editing and rendering of reactive models. The result is an artifact that replicates and extends the spreadsheet computing experience.

The ZenSheet project is far from complete. As already mentioned, we aim to extend the type system with sum types, recursive types, and parametric type polymorphism. Moreover, we are now ready to explore how to abstract and generalize effectful user interactions. As presented here, model mutation is almost exclusively the result of some user interaction represented by a Lilly action. We separated actions from expressions precisely to match the quasi-pure nature of the spreadsheet core. We say "quasi-pure" because stochastic functions, like RANDOM, and time functions, like NOW, are obviously not pure. One of the toughest challenges ahead is

finding how to best add computational effects to Lilly. We aim to make such effects reasonably controllable and predictable in a parallel-reactive environment. That's already a tall order and the subject of ongoing research. On top of all that, we also need to preserve the intuitive and user-friendly nature of spreadsheet computing as much as possible.

Last, we wish to test ZenSheet in an education setting, as we believe it will be suitable to complement the teaching of STEM subjects while introducing students to computational thinking and sound programming practices.

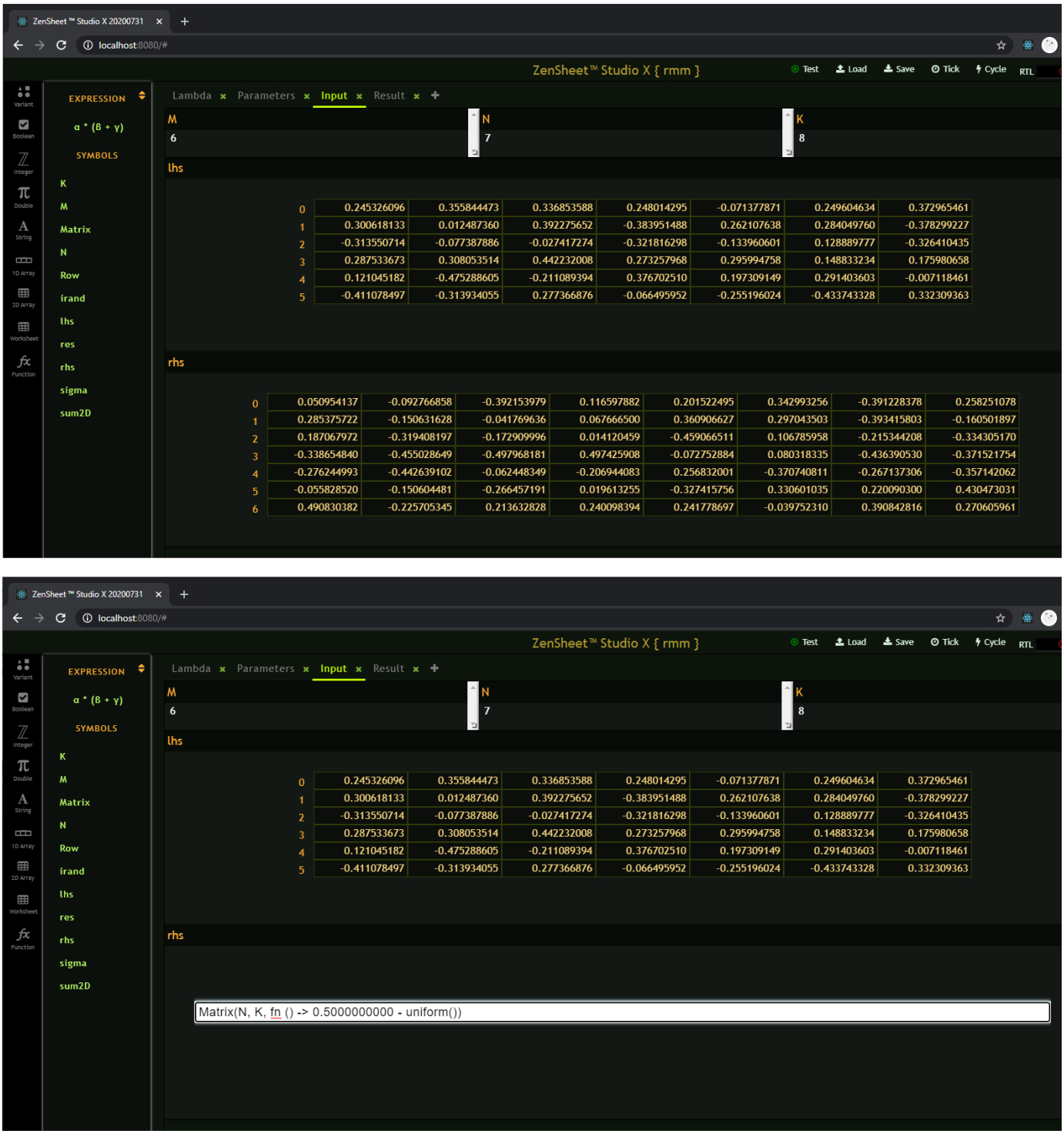


Fig. 1. rmm model – matrix multiplication input matrices: *lhs* is being displayed (top) and *rhs* is being edited (bottom)

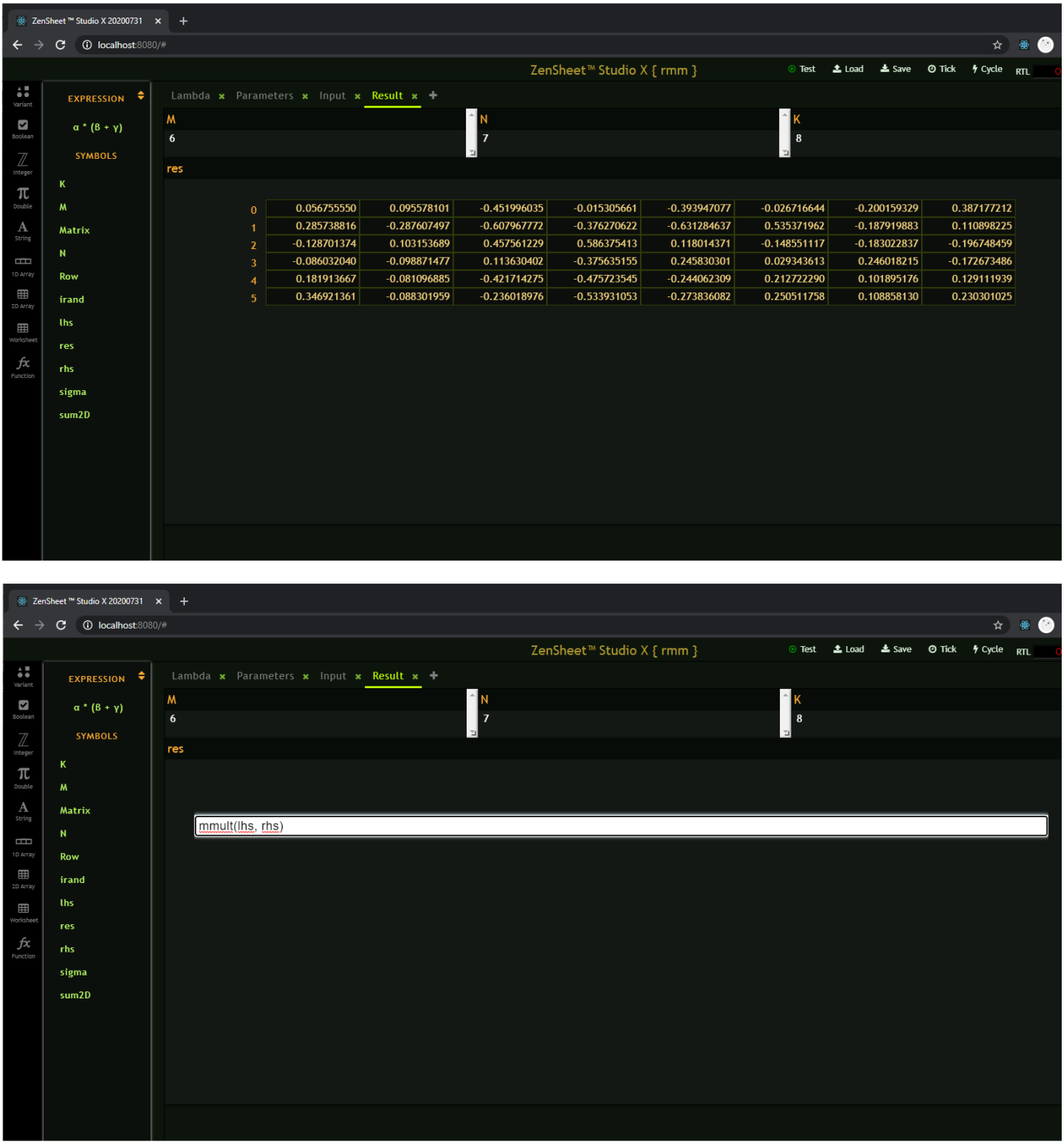


Fig. 2. rmm model – matrix multiplication result matrix in display mode (top) and editing mode (bottom)

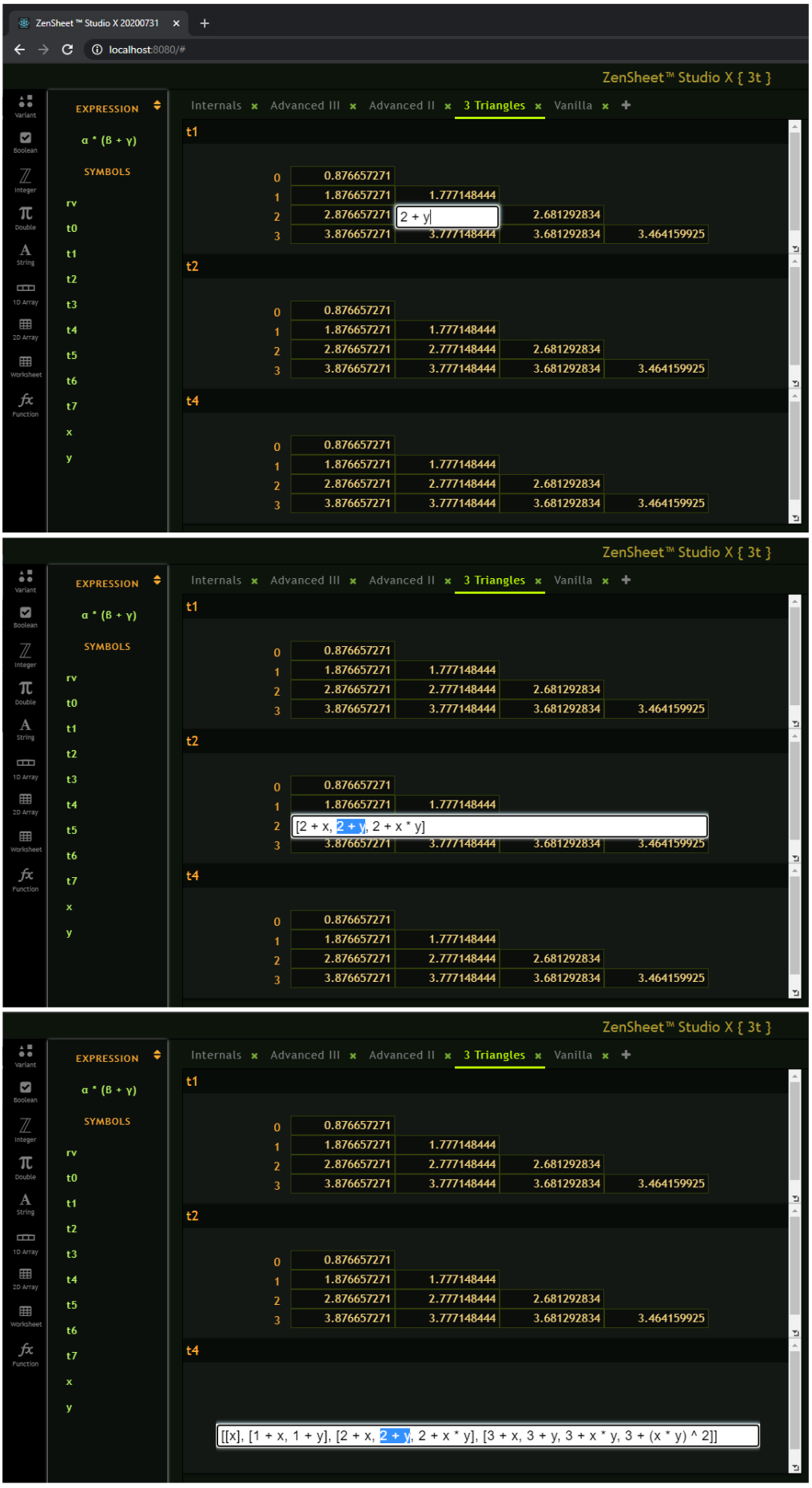
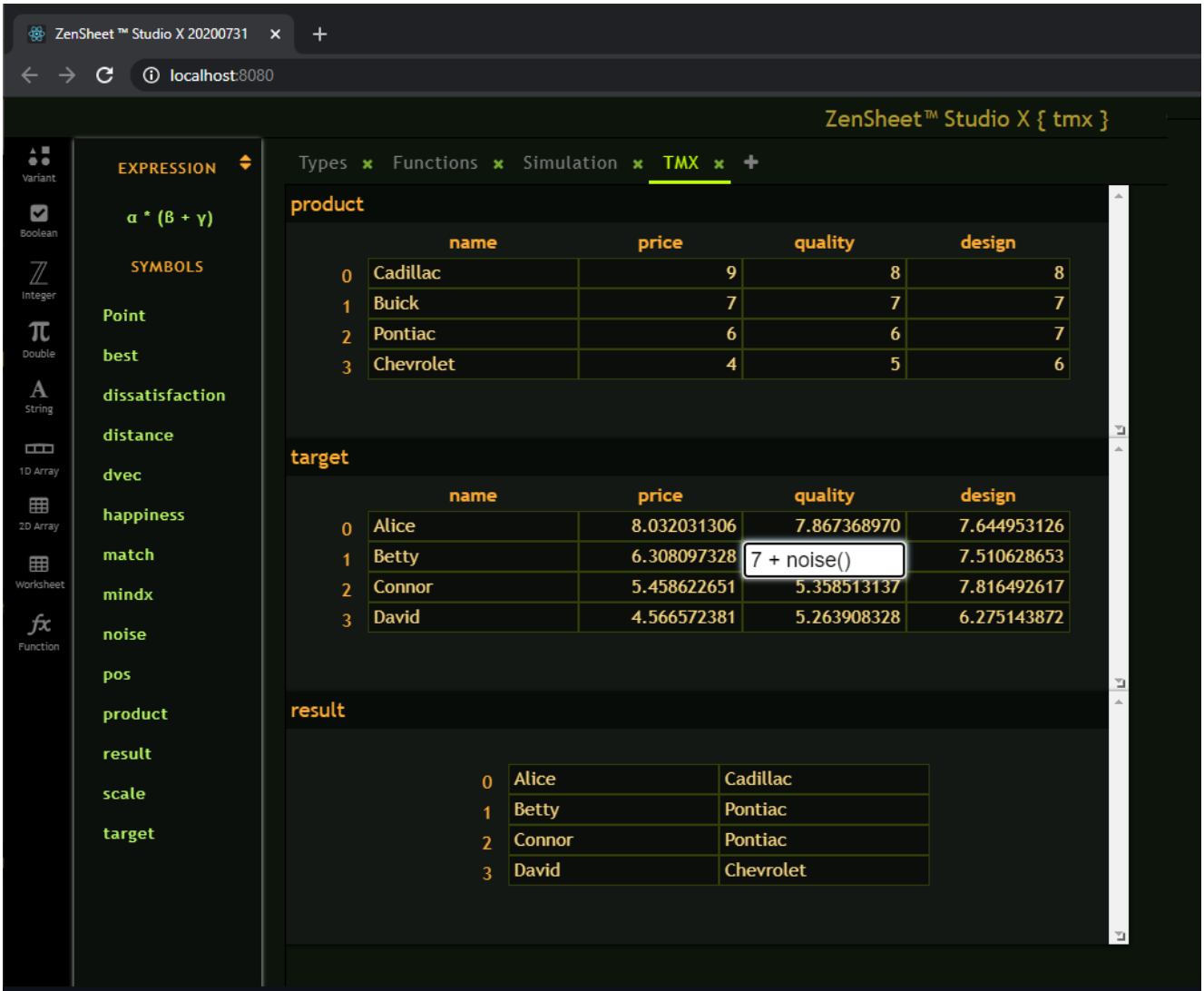


Fig. 3. 3t model – three triangular structures with mutable lazy cells at different nesting level



tmx-code.sym

```
type Point := struct {
  string name;
  lazy double price;
  lazy double quality;
  lazy double design;
};

array[] => Point product := [
  Point("Cadillac", '9', '8', '8'),
  Pint("Buick", '7', '7', '7'),
  Point("Chevrolet", '4', '5', '6')
];

array[] => Point target := [
  Point("Alice", '8 + noise()', '8 + noise()', '8 + noise()'),
  Point("Betty", '6 + noise()', '7 + noise()', '9 + noise()'),
  Point("Charlie", '5 + noise()', '4 + noise()', '8 + noise()'),
  Point("David", '5 + noise()', '6 + noise()', '6 + noise()')
];

fun (Point p, Point q) => double -> distance := fn (Point p, Point q) ->
  sqrt((p.price - q.price) ^ 2 + (p.quality - q.quality) ^ 2 + (p.design - q.design) ^ 2);
```

Fig. 4. tmx model – targeted marketing stochastic simulation

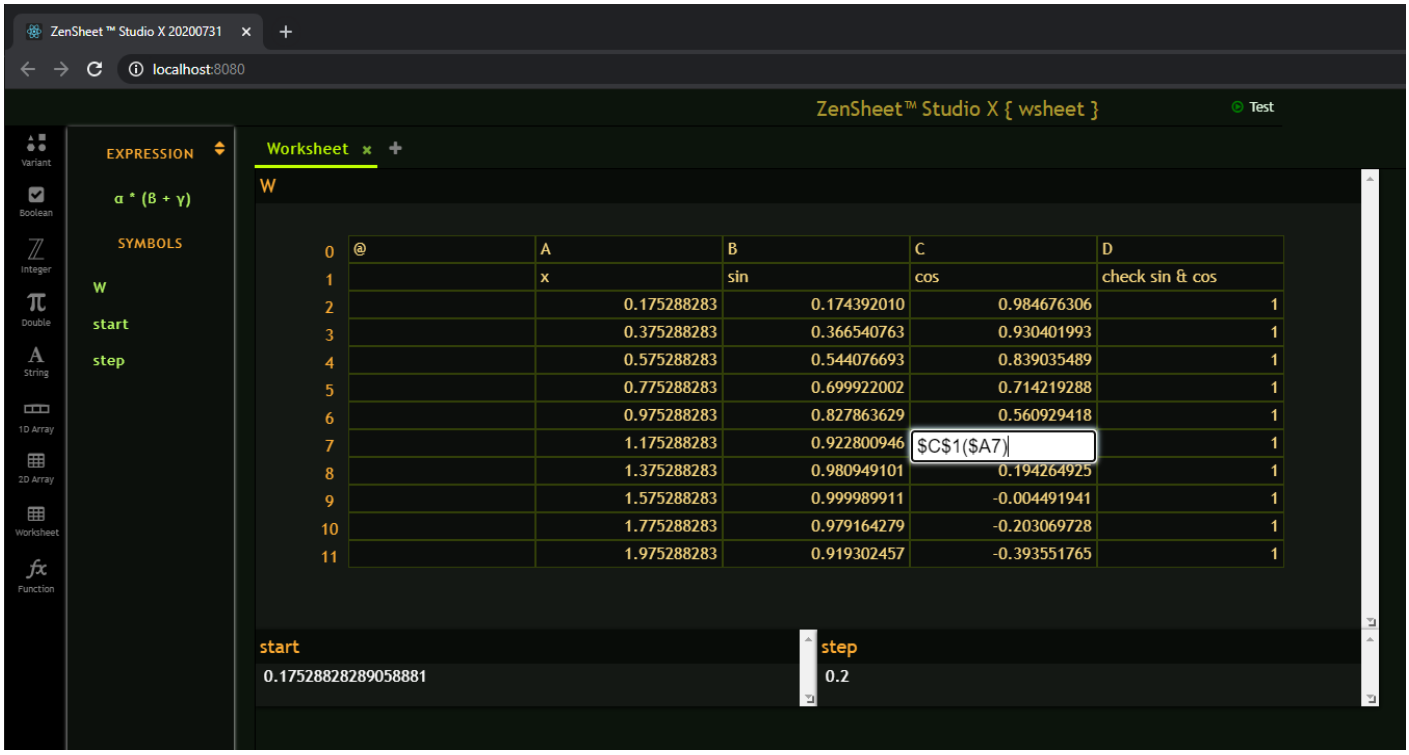


Fig. 5. wsheet model – traditional worksheet behavior in a 2D array of lazy variants

ACKNOWLEDGMENT

We wish to thank faculty members at Yale University, University of Oregon, Princeton University, Universidad Simón Bolívar, and Universitá Federico II, as well as other researchers and students we had the pleasure to meet at various conferences. We have decided to opt for a seemingly pointless "anonymous recipients" acknowledgment because there are too many of them and we are wary of seemingly implying any endorsement of the ideas here expressed. They know who they are and we want show our *deep gratitude* for their encouragement and suggestions in our 11 years journey so far.

REFERENCES

[1] F. Hermans, "Keynote: How to teach programming and other things?," *https://www.youtube.com/watch?v=UJxXgugvXmE*, 2018. .

[2] S. P. Jones, A. Blackwell, and M. Burnett, "A user-centred approach to functions in Excel," in *ACM SIGPLAN Notices*, 2003, vol. 38, no. 9, pp. 165–176, doi: 10.1145/944746.944721.

[3] R. Abraham, M. Burnett, and M. Erwig, "Spreadsheet Programming," in Wiley *Encyclopedia of Computer Science and Engineering*, 2009, pp. 1–10.

[4] S. P. Jones, M. Burnett, and A. Blackwell, "Spreadsheets : functional programming for the masses." https://www.slideshare.net/kfrdbs/peyton-jones.

[5] Microsoft Research, "Future of Spreadsheeting," 2019. https://www.microsoft.com/en-us/research/video/future-of-spreadsheeting/.

[6] Microsoft Research, "LAMBDA," 2020. https://techcommunity.microsoft.com/t5/excel-blog/announcing-lambda-turn-excel-formulas-into-custom-functions/ba-p/1925546.

[7] M. McCutchen, J. Borghouts, A. D. Gordon, and S. P. Jones, "Elastic Sheet-Defined Functions : Generalising Spreadsheet Functions to Variable-Size Input Arrays ∗," vol. 1, no. March, 2018.

[8] M. Figuera, "ZenSheet Studio: A Spreadsheet-Inspired Environment for Reactive Computing," in *SPLASH Companion 2017 - Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, Oct. 2017, pp. 33–35, doi: 10.1145/3135932.3135949.

[9] E. Alda and M. Figuera, "ZenSheet: a live programming environment for reactive computing," 2017. https://2017.splashcon.org/details/live-2017/5/ZenSheet-a-live-programming-environment-for-reactive-computing.

[10] Microsoft Research, "Preview of Dynamic Arrays in Excel," 2018. https://techcommunity.microsoft.com/t5/excel-blog/preview-of-dynamic-arrays-in-excel/ba-p/252944.

[11] J. G. Siek and W. Taha, "Gradual typing for objects," in *ECOOP'07*, 2007, pp. 2–27.

[12] E. Alda and J. Lopéz, "Lambda Days 2020," 2020. https://www.youtube.com/watch?v=mJa0_gKE6xo.