Towards an Operational Semantics for a Generalized Spreadsheet Core

Enzo Alda [1][0000-0002-4663-6261]

[1] Lakebolt Research, Quincy MA 02169, USA
lncs@springer.com

Abstract. We present an operational semantics for a subset of a programming language that serves as the foundation for a reactive computing environment (ZenSheet) that generalizes spreadsheets with functional abstraction and composable containers, while supporting static typing alongside the dynamic typing (unitype) approach found in commercial spreadsheets.

Keywords: Reactive Computing, Functional Programming, Spreadsheets.

1 Introduction

As recognized by computer science researchers, though not yet by most people, spreadsheets are computing environments and spreadsheet modeling is a form of computer programming [SPJ 2003] [Hermans 20??]. VisiCalc, introduced in 1979, is credited as the killer app that ignited the personal computing revolution. Over 40 years later, spreadsheets are still going strong.

Spreadsheets combine an equation based functional-reactive computing paradigm with an intuitive visual interface. Unfortunately, the elegance of that equational paradigm was stained by the addition of various programming adjuncts, like macros and scripting languages, not designed as generalizations of the language of formulae. That was a partial, and inadequate, response to spreadsheet shortcomings that still persist:

- Lack of functional abstraction in worksheet formulae, with some recent exceptions
- The implied type-system is very poor and lacks support for static type checking
- Computation models are entangled with the view (presentation) of the same.

The ZenSheet project [LIVE 2017] started as an expedition exploring the possibility of turning spreadsheets into IDEs for a suitable class of general-purpose programming languages, eliminating the need for extraneous programming adjuncts.

The project focused on three questions:

- What are spreadsheets like from a programming language perspective?
- Is it possible to extend spreadsheets with modern programming language concepts?
- Will the result be amenable to spreadsheet practitioners and software engineers?

2 Analysis of the Traditional Spreadsheet Core

We often use the word "core" as short for "spreadsheet core", which can be loosely described as the part of a spreadsheet environment consisting of the worksheets and formulae (including values) contained in them. All the questions above are predicated on the core because it is the elegance of the core we want to extend. Taking a look at formulae in traditional spreadsheets, we can infer the following abstract grammar:

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

The productions above correspond to literal constants (including null values), function application, single cell reference, and cell range reference. Note that all forms of cell reference use A1 notation, optionally prefixed with a worksheet label, considered a symbol for practical purposes, followed by an exclamation point. Deliberately, we don't include aliases for worksheet regions among valid cell references because they are a poor substitute for properly defined named variables.

We consider operations on worksheets, which we call actions, an integral part of the spreadsheet computing paradigm. Limiting ourselves to the core, these operations can only be performed interactively by users, since (fortunately) formulae are essentially devoid of side effects other than, for instance, changing the internal state of random number generators. Here we show the abstract grammar of actions:

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

The actions correspond to adding and removing worksheets, editing a cell, performing copy and paste operations (of formulae or values), and insert/delete operations on worksheet rows and columns.

{ conclude section with the following points … }

The need to automate actions performed by users was the trigger for the addition of the imperative programming adjuncts that did not honor the language of formulae in the core.

{ Explain the advantages of our language-centric approach }

{ Introduce the following section }

3 Lilly Design Considerations

{ This section needs a LOT of work }

… In [ICICT 2021] we describe our proposed generalization of the spreadsheet core:

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>

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’

Actions

ZSA.1) A → **type** <symbol> = T;

ZSA.2) A → T <symbol> := E;

ZSA.3) A → E := E;

Listing 3: abstract syntax of Lilly

Explain how this paper is a continuation of
[ICICT 2021] https://ieeexplore.ieee.org/document/9476942

[ICICT 2021] informally describes how Lilly covers the functionality of the spreadsheet core. Given ZenSheet's language-centric approach, it is only fair to demand a more precise semantic definition of Lilly. This paper is a step in that direction.

{ clean, consolidate and remove as much as possible for brevity }

Lilly easier to understand design choices => language

major inspiration: spreadsheets
A1 notation is to be considered harmful
The only symbols defined by the user are the worksheets names (actually labels)
Worksheets effectively are 2D regions of memory and users are the allocators

unification of spreadsheets and general-purpose computing environments
adopting widely accepted principles of programming language design
– [LIVE 2017] [ICICT 2021]

{ explain this: ZenSheet has been radical in following a language-centric approach }

Two implementations: from interpreter to transpiler - from codename Peano to Lilly

{ mention how spreadsheets have evolved since LIVE 2017 }

Relatively recently Microsoft added ...
- Dynamic arrays (2019)
- Lambda expressions (2020)

Mention other work:

- Conal and Hudak FRP
- Haxcell
- Peter Sestoft – Corecalc and Funcalc
- Lustre
- Clean? Others?

The state of a spreadsheet is described by its worksheets and the formulae contained in them.

Worksheets are the only data structures supported.

[SPJ 2003]

Interaction is of the essence. Imperative programming adjuncts evolved from the need to automate said interactions.

Our view of consider user interaction of the essence
cells formulae -> values
Define RV, CV and the =RV=> transform
Changing a cell ➔assignment

Separating expressions from actions

How Lilly extends Christopher Strachey [Strachey 1967]
l-value, c-value, r-value
modified eval rules

compute cycle semantics - consistent view
model evolution - mutation
discretized synchronous reactive computing

A1 notation had to be honored
worksheets: array[,] => lazy var
support for A1 notation, including copy & paste values and formulae

static typing

ZenSheet

compute cycle semantics - consistent view
definition blocks
model healing
circularity

MS-Excel dynamic circular definition

4 Zilly

We assume a fictitious (idealized) system that can handle arbitrarily large integer numbers and has capacity for an unlimited, but finite, number of variables. The type int (integer) is the only basic type available. The syntax shown here is to be taken as a representation of an abstract syntax: it is not the concrete syntax of Zilly - which in turn is a small subset of Lilly's syntax - but has a direct correspondence to it.

4.1 Syntax

Types:

$$T \Rightarrow int \tag{1}$$

$$T \Rightarrow (T => T) \tag{2}$$

$$T \Rightarrow lazy\ T \tag{3}$$

Expressions:

$$E \Rightarrow < integer > \tag{4}$$

$$E \Rightarrow < symbol > \tag{5}$$

$$E \Rightarrow\ T\ < symbol\ > => T \rightarrow E \tag{6}$$

$$E \Rightarrow\ E\ E \tag{7}$$

$$E \Rightarrow\ if(E,E,E) \tag{8}$$

$$E \Rightarrow\ 'E' \tag{9}$$

Actions:

$$A \Rightarrow\ T\ < symbol\ > := E; \tag{10}$$

$$A \Rightarrow < symbol\ > := E; \tag{11}$$

4.2 Zilly Subsets

Though Zilly appears fairly small, there are three subsets worth mentioning:

- Eliminating (11) we ditch imperative programming.
- Eliminating (3) and (9) we let go of lazy evaluation. – Bye to reactive behavior.
- Eliminating (3), (9), and (11) … we still have a Turing-complete language!

{ Mention (reiterate?) why (11) follows from our design goal of modeling user interaction and the evolution of the computing model. }

The conditional expression if in rule (8), which has exactly the same syntax as the IF function in MS-Excel, is a special form: the first expression, which must have r-type *int*, is evaluated first and, depending on the resulting value, either the second or the third expression, which must have the same r-type, is evaluated to yield the final result of the entire conditional expression.

The derivation rules (6) and (7) correspond to functional abstraction and functional application respectively. Given a functional abstraction of the form

$$\theta\ \upsilon\ {->}\ \varphi \tag{12}$$

and the following typing judgment implication

$$\Gamma, \upsilon{:}\,\theta\ \Rightarrow\ \Gamma, \upsilon{:}\,\theta, \varphi{:}\,\tau \tag{13}$$

the r-type of the functional abstraction is $\theta\ {=>}\ \tau$

$$\Gamma, \theta\ \upsilon\ {->}\ \varphi{:}\,\theta\ {=>}\ \tau \tag{14}$$

Moreover, given these two typing judgments

$$\Gamma, \lambda{:}\,\theta\ {=>}\ \tau\ \wedge\ \Gamma, \mu{:}\,\theta \tag{15}$$

we can conclude that the r-type of the functional application $\lambda\ \mu$ is τ

$$\Gamma, \lambda\ \mu{:}\,\tau \tag{16}$$

A complete list of typing judgments for expressions is provided in Appendix T.

4.3 Expression Evaluation

$$\eta\ = RV\ {=>}\ \eta \tag{17}$$

$$\upsilon\ = RV\ {=>}\ RV(CV(\upsilon)) \tag{18}$$

$$\theta\ \upsilon\ {->}\ \varphi\ = RV\ {=>}\ \theta\ \upsilon\ {->}\ \varphi \tag{19}$$

$$\lambda\ \mu\ = RV\ {=>}\ RV(RV(\lambda)\ RV(\mu)) \tag{20}$$

$$(\theta\ \upsilon\ {->}\ \varphi)(\alpha)\ = RV\ {=>}\ RV(\varphi[[\upsilon/\alpha]]) \tag{21}$$

$$if(\delta, \varphi, \psi)\ = RV\ {=>}\ RV(if(RV(\delta), \varphi, \psi)) \tag{22}$$

$$if(0, \varphi, \psi)\ = RV\ {=>}\ RV(\psi) \tag{23}$$

$$if(\eta, \varphi, \psi)\ = RV\ {=>}\ RV(\varphi) \quad \{\ where\ \eta \neq 0\ \} \tag{24}$$

$$'\varphi'\ = RV\ {=>}\ \varphi \tag{25}$$

4.4 Predefined Zilly Functions

We stipulate that the Zilly VM offers the following functions.

$$::lt := int\ x -> int\ y -> x < y; \tag{26}$$

$$::minus := int\ x -> int\ y -> x - y; \tag{27}$$

$$::random := int\ k -> \mathbf{random}(k); \tag{28}$$

$$::formula := \theta'\ v -> \mathbf{CV}(v); \tag{29}$$

All the "magic" essence is in orange. The operators '-' and '<' are the infix binary arithmetic operators for subtraction and precedence. Given a positive, non-zero, *int* value k, the ***random*** function returns a random int number r, $0 \leq r < k$. We also stipulate that ***random****(0)* always returns 0 and ***random****(n)* = -***random****(-n)* for negative *n*.

4.5 Compute Cycle Semantic

Actions change the state of the system. If correctly typed, action (10) defines a new variable and action (11) changes the value of a variable.

{ define compute cycle }

{ show compute cycle eval algorithm in Appendix }

Property: memoized values are consistent with formulae in the computation model.

Mutually recursive definitions

What about mutually recursive definitions? They are also handled.

{ briefly explain the two approaches we implemented to do so }

5 Generalization to Lilly

{ give an informal overview of the generalization of the above to: }

- multidimensional and dynamic arrays
- product types: tuples and structs
- more basic types and the addition (gradual typing) of the unlabeled variant type *var*

{ provide insight into the implications of having containers and lazy evaluation }
{ give a semi-formal proof of this: } given our definition of *lazy T* and *var*, we have:

$$var \ = lazy\ var \tag{30}$$

6 “Outro”

{ future directions – probably omit for brevity or move details to appendix }

6.1 mixed eager-lazy evaluation

The quoting of expressions mechanism makes it possible to defer evaluation The eager evaluation mechanism makes it possible to override the deferral. The evaluation process allows arbitrary use of these mechanisms on different parts of an expression and even nesting them without restrictions, making it possible to switch between eager and lazy evaluation at will. To specify with precision the method herein described, consider the following syntax of expressions:
E -> <int>
E -> <symbol>
E -> E(E, …, E)
E -> 'E'
E -> $(E)

We extend the evaluation process according to the rules below:

Eager evaluation (RV)
RV(<int>) = <int>
RV(<symbol>) = RV(CV(<symbol>))
RV(E[f](E[1], …, E[k])) = RV(E[f])(RV(E[1]), …, RV(E[k]))
RV('E') = XV(E)
RV($(E)) = RV(E)

Deferred evaluation (XV)
XV(<int>) = <int>
XV(<symbol>) = <symbol>
XV(E[f](E[1], …, E[k])) = XV(E[f])(XV(E[1]), …, XV(E[k]))

XV('E') = ‘XV(E)’
XV($(E)) = RV(E)

Also ➔ full evaluation and lazy* T

6.2 lazy propagation

DX. Laziness propagation
Lazy propagation is a transform that “cures” function calls that would otherwise fail type checking. To illustrate laziness propagation with an example, let’s consider the following definitions:
:: f := fn(int x, int y) => int -> … ;
:: a := 3;
:: b := 4;
:: c := 6;
:: d := 7;
Function f’s body is not material to the example. Let’s pretend we try to call f as follows:
f(a + b, ‘c + d’)
In principle, the call above is not allowed because function f expects an integer value as the second argument, not an unreduced expression. The following calls, however, would be fine:
‘f(7, c + d)’
‘f(a + b, c + d)’
Instead of deferring the evaluation of c + d, which implies the evaluation of c and d as well, we can defer the call expression in its entirety. In the first case we evaluate the argument passed as the x parameter before quoting the call expression (i.e., propagating laziness), in the second we don’t.
When enabled by the user’s preferences, laziness propagation will perform one of the transformations above automatically. As is always the case, the result of evaluating the quoted expression is the enclosed expression, still unreduced:
‘f(7, c + d)’ =RV=> f(7, c + d)
‘f(a + b, c + d)’ =RV=> f(a + b, c + d)
Note how both transformations above result in an expression of type lazy int, which may in turn cause further propagation of laziness.
Analogous transformations apply when using operators. This should be unsurprising given that operators are nothing but “syntactic sugar” for calling functions. For instance, function f in the example above may in fact be a function that calculates the product of its arguments. If so, the Lilly VM could implement the “star” (*) operator by converting each use into the corresponding function call, i.e.:
x * y is de-sugared (converted) to f(x, y)
Looking at the case at hand from the perspective of using operators, and assuming there is no support for laziness propagation, we have:
(a + b) * ‘c + d’ =RV=> ERROR (fails type checking for the same reason stated above)

There are two possible transformations. If post eval propagation is enabled, the arguments are evaluated before laziness is propagated:
(a + b) * ‘c + d’ =RV=> 7 * (c + d)
Note that there is no need to enclose the expression ‘c + d’ in parenthesis on the left of the transformation rule above: the quotes themselves serve as parenthesis. However, to preserve proper parsing after removing the quotes, the expression c + d must be enclosed in parenthesis on the right.
If pre eval propagation is enabled, laziness is propagated before the arguments are evaluated:
(a + b) * ‘c + d’ =RV=> (a + b) * (c + d)
Looking at the function call alone in the examples above, it is not clear what added value laziness propagation provides. But when we look at the use of an expression in the context of a variable definition, the benefits become clear. For instance, in the following definition:
lazy int z := (a + b) * ‘c + d’;
the Lilly VM will perform laziness propagation while evaluating the initializer. The result then becomes the CVALUE of variable z, according to the assignment rules. Depending on the propagation flavor enabled, we end up with one of the following states for z:

formula(z) ==> 7 * (c + d)
formula(z) ==> (a + b) * (c + d)
In the first case (post eval propagation) z is a lazy variable that only depends on c and d. In the second case, z depends on all the other variables. The first case is the default setting: it has the obvious advantage of freezing the result of evaluating part of the expression, a + b in this case, at the time of initialization. What we have here is a “toy example”, but there are situations where we need to perform a costly computation to determine part of an expression (e.g., a model coefficient) which we know won’t need to be changed after it is determined. In this case, freezing that result and embedding it in a formula can have a very significant impact on performance. The second case can have some advantages in the fields of symbolic processing and generative programming.

7 Appendix A, B, … TBA

7.1 Lilly Code Examples

Shows the consequence of the double evaluation mentioned above.

```
:: txx := fn(lazy lazy int x) => lazy int
    ->  x * x - 4 * x + 3;
:: a := 6;
:: b := 7

txx(a + b) ==> 120
txx('a + b') ==> 120
txx(''a + b'') ==> (a + b) * (a + b) - 4 * (a + b) + 3
```

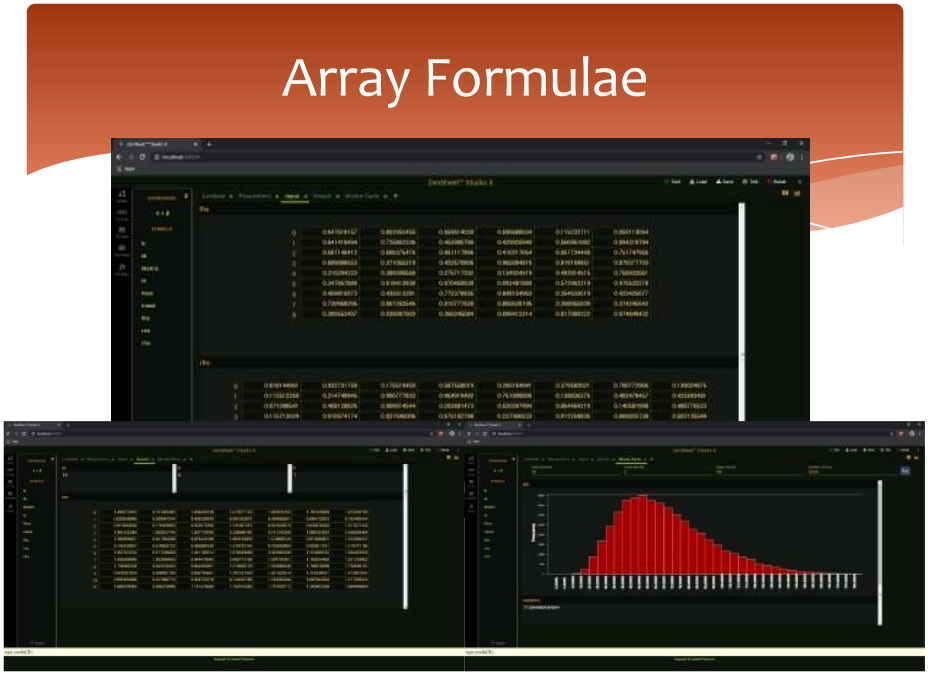
{ … add a few good ones later … }

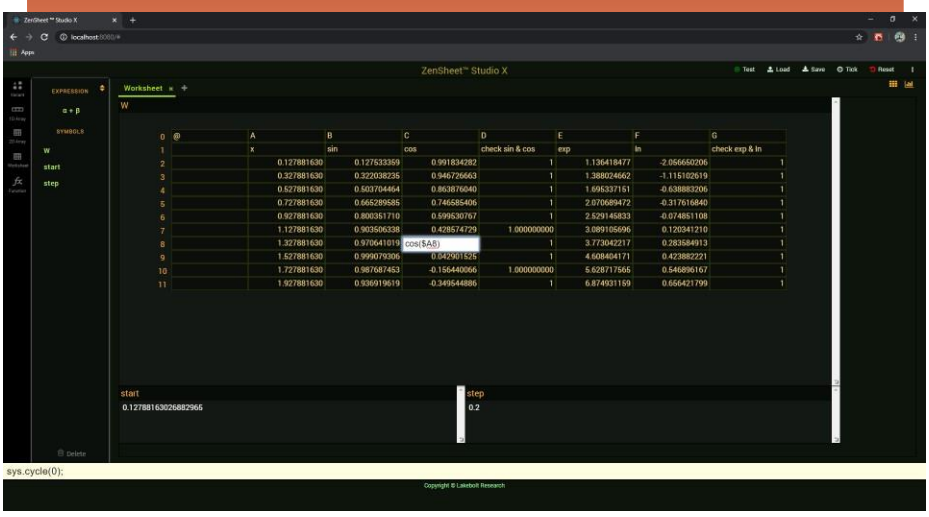
7.2 Compute Cycle

```
{ convert to more legible pseudo-code }
```

```
bool LazyVar::reduce(evn_t cycle) const
{
  // Realization of the two-count ev-cycle algorithm ...
  if (evn_comp_ == cycle) {
    if (evn_last_ == cycle) {
      ++statistics.hits_;
      return false;
    } else {
      evn_comp_ = cycle;
      reduced_ = ABT::Invalid("#REF!", "circular ref");
      ++statistics.invalid_;
      return true;
    }
  } else {
    evn_comp_ = cycle;
    ++statistics.folds_;
    ABT previous = reduced_;
    reduced_ = optimized_.reval(cycle);
    evn_last_ = cycle;
    if (reduced_.is_invalid()) {
      ++statistics.invalid_;
    }
    previous_ = previous;
    if (previous == reduced_) {
      return false;
    }
  }
  return true;
}
```

7.3 Some actual model screenshots, just in case (?)





8 References (TBA)

1. Author, F.: Article title. Journal 2(5), 99–110 (2016).
2. Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016).
3. Author, F., Author, S., Author, T.: Book title. 2nd edn. Publisher, Location (1999).
4. Author, F.: Contribution title. In: 9th International Proceedings on Proceedings, pp. 1–2. Publisher, Location (2010).
5. LNCS Homepage, http://www.springer.com/lncs, last accessed 2016/11/21.