

Towards an Operational Semantics for a Generalized Spreadsheet Core

Enzo Alda¹[0000-0002-4663-6261] and Daniel Pinto²[0000-0002-4663-6261]

¹ Lakebolt Research, Quincy, Massachusetts, USA

enzo@ieee.org | <https://www.lakebolt.com/>

² Universidad Simón Bolívar, Valle de Sartenejas, Miranda, Venezuela

15-11139@usb.ve | <http://www.usb.ve/>

Abstract. We present an operational semantics for a subset of Lilly - a programming language that mimics spreadsheet behavior. Lilly is the pillar of a reactive computing environment that extends spreadsheets with functional abstraction, composable containers, and a type system that augments the dynamic typing approach found in commercial spreadsheets with optional static typing. The chosen subset, named Zilly, captures the fundamental features that make Lilly capable of replicating spreadsheet behavior - a key step towards the formalization of Lilly.

Keywords: Reactive Computing, Spreadsheet Semantics.

1 Note to reviewers

This paper (in progress) is partly a continuation of one presented at ICICT 2021, titled "Towards Wide-Spectrum Spreadsheet Computing". Our submission is complemented by a repository containing:

- A copy of the ICICT 2021 paper (<https://ieeexplore.ieee.org/document/9476942>)
- An artifact, written in Haskell, that supports our formalization work

The repository is ready at <https://github.com/Liondance/TFP-2024>

The artifact embodies our formalization of a small, yet relevant, subset of the Lilly language. We are aware that the deadline has long passed. No matter what, we plan to attend the symposium and will continue working on this paper afterwards. Advancing a formal specification of our spreadsheet generalization is an important and long overdue step in our research.

Thank you,

- The Authors

2 Introduction

Spreadsheet computing turns 45 this year: VisiCalc was released on October 17, 1979. Spreadsheets are the killer app that ignited the personal computing revolution [BBC][Bricklin][BYTE] and are still going strong: it is not an exaggeration to say that spreadsheets are the most popular computing environment in the world. [SPJ][Sestoft] Spreadsheet modeling is a form of programming, as recognized by computer science researchers. For instance, in [SPJ 2003]:

“It may seem odd to describe a spreadsheet as a programming language. Indeed, one of the great merits of spreadsheets is that users need not think of themselves as doing “programming”, let alone functional programming—rather, they simply “write formulae” or “build a model”. However, one can imagine printing the cells of a spreadsheet in textual form, like this:

```
A1 = 3
A2 = A1-32
A3 = A2 * 5/9
```

and then it plainly is a (functional) program.”

Spreadsheets realize an equation-based computing paradigm in a live programming environment featuring an intuitive visual interface. Unfortunately, the elegance of that paradigm, still alive in the core of commercial spreadsheets, was stained by the addition of extraneous elements - not without cause.

If you look at a spreadsheet from a static perspective then, yes, it is a pure functional program. Changes made to it are then to be seen as “changing the code”, i.e., “editing the program”. That is a consistent point of view. Yet, there is another one, that is also consistent: changes made to the spreadsheet are runtime state mutations, only ... they are performed by the user, not the program!

The second perspective feels bizarre, and even abhorrent, to many. We are aware, and hope our work will convince you that this path is worth exploring.

This paper is a first step towards formalizing the Lilly language, the pillar of ZenSheet [ICIT 2021] - an experimental computing environment aiming to unify spreadsheets and programming languages. Besides getting helpful feedback on the formalization itself, we believe this “exercise” will give us more insight into the alternatives, still being explored, for handling effectful computations.

Chapters still being worked, please jump to chapter 5 to see some "beef".

3 What’s wrong with commercial spreadsheets?

TBW: (1 page) summarize problems in the traditional core:
Lambda Days (2017 - 2020), LIVE (2017), and ICIT (2021)

4 Generalizing the spreadsheet core

- changing a cell => editing the program
- copy and paste => editing the program
- adding a worksheet => editing the program

Yet, there is another way to look at this, that rings true to programmers:

- changing a cell => recomputing (executing) with new data, or editing code
- copy and paste => performing a multiple assignment, or editing the code
- adding a worksheet => creating a new variable, or editing the code

The operations above could only be performed by hand in early spreadsheets. We should keep in mind that spreadsheets are a live programming environments: the lines between editing, compiling, and running are blurred. This makes choosing one of the two interpretations very debatable from a theoretical perspective.

But, from a historical and practical perspective, the evidence overwhelmingly favors the second interpretation: the need to automate transformations that could only be performed manually was the driver for adding macros and then hastily “plopping” imperative programming languages on top of the spreadsheet core. It was a quick and practical way to solve the problem, but also a “dirty” one from a programming language design perspective.

Choosing the second perspective is the distinctive characteristic of the Zen-Sheet project: an attempt to unify spreadsheets and programming languages. Everything stems from taking that perspective and following well-known principles of programming language design.

by generalizing the spreadsheet core as opposed to adding extraneous elements to it. Is a language-centric approach.

TBW: (2 pages)

- Explain the main idea and why we believe this is the way
- Summarize ideas previously presented
- Why are we formalizing Zilly instead of Lilly?
 - to keep this short
 - to learn ourselves (first time doing this)
 - to make this "exercise" more manageable, yet ...
 - ... minimally SUFFICIENT to show virtues and shortcomings!

–

***** <i>Definitions!</i> <i>Definitions!</i> *****

5 Zilly

We assume a fictitious and idealized system that has capacity for an unlimited, but finite, number of variables. The type Z (integer) is the only basic type available. Each variable can hold an arbitrarily large integer number or an arbitrarily long expression. Global variables are mutable.

5.1 Abstract syntax

Types

$$t ::= Z \tag{1}$$

$$t ::= Fun(t_p, t_r) \tag{2}$$

$$t ::= Lazy(t_b) \tag{3}$$

Expressions

$$e ::= i \quad (\text{where } i = 0 \mid 1 \mid 2 \mid \dots) \tag{4}$$

$$e ::= x \quad (\text{where } x \text{ is a symbol}) \tag{5}$$

$$e ::= Lambda(x : t, e_x) \tag{6}$$

$$e ::= Apply(e_f, e_a) \tag{7}$$

$$e ::= If(e_c, e_t, e_f) \tag{8}$$

$$e ::= Defer(e_x) \tag{9}$$

$$e ::= Less(e_x, e_y) \tag{10}$$

$$e ::= Minus(e_x, e_y) \tag{11}$$

$$e ::= Random(e_x) \tag{12}$$

$$e ::= Formula(e_x) \tag{13}$$

Statements

$$a ::= Define(x : t, e); \tag{14}$$

$$a ::= Assign(x, e); \tag{15}$$

Zilly subsets: Though small, Zilly has three subsets worth mentioning:

- * Eliminating (15) precludes an imperative programming style
- * Eliminating (3) and (9) precludes lazy evaluation and reactive behavior
- * Eliminating (3), (9), and (15) still leaves a Turing-complete language

5.2 Natural semantics of Zilly expressions

$$\begin{array}{c}
\overline{\Gamma \vdash i \Downarrow i} \quad \overline{\Gamma \vdash \text{Lambda}(x : t, e) \Downarrow \text{Lambda}(x : t, e)} \\
\\
\frac{\Gamma \vdash x \Downarrow_c e_c \quad e_c \Downarrow e_r}{\Gamma \vdash x \Downarrow e_r} \\
\\
\frac{\Gamma \vdash e_1 \Downarrow \text{Lambda}(x, e_x) \quad e_2 \Downarrow e_a \quad e_x[x/e_a] \Downarrow e_r}{\Gamma \vdash \text{Apply}(e_1, e_2) \Downarrow e_r} \\
\\
\frac{\Gamma \vdash e_1 \Downarrow 0 \quad e_3 \Downarrow e_r}{\Gamma \vdash \text{If}(e_1, e_2, e_3) \Downarrow e_r} \quad \frac{\Gamma \vdash e_1 \Downarrow i \quad i \neq 0 \quad e_2 \Downarrow e_r}{\Gamma \vdash \text{If}(e_1, e_2, e_3) \Downarrow e_r} \\
\\
\overline{\Gamma \vdash \text{Defer}(e) \Downarrow e} \\
\\
\frac{\Gamma \vdash e_l \Downarrow i_l \quad e_r \Downarrow i_r}{\Gamma \vdash \text{Less}(e_l, e_r) \Downarrow \textcolor{red}{I}(i_l < i_r)} \quad \frac{\Gamma \vdash e_l \Downarrow i_l \quad e_r \Downarrow i_r}{\Gamma \vdash \text{Minus}(e_l, e_r) \Downarrow i_l - i_r} \\
\\
\frac{\Gamma \vdash e \Downarrow i \quad i < 1}{\Gamma \vdash \text{Random}(e) \Downarrow 0} \quad \frac{\Gamma \vdash e \Downarrow i \quad 1 \leq i}{\Gamma \vdash \text{Random}(e) \Downarrow z \textcolor{red}{mod} i} \quad | \text{ where } z \in \mathbb{Z} \\
\\
\frac{\Gamma \vdash e \Downarrow_c e_f}{\Gamma \vdash \text{Formula}(e) \Downarrow e_f}
\end{array}$$

Fig.1

Fig 1 remarks, including a few obvious ones:

- The symbol \Downarrow represents the r-value reduction and \Downarrow_c the c-value reduction.
- Integer constants and lambda expressions are already reduced.
- To obtain the value (r-value) of a variable used as an expression - e.g. on the right hand side of an assignment or as an argument in a function application - we obtain the *content* (c-value) of the variable, an expression e_c , and evaluate e_c in the given context.
- Function parameters are immutable - by construction the body of a lambda can't contain actions - but have a c-value: they are variables. The same is true for symbols hypothetically defined by *let* expressions - *let* expressions can be transformed to function calls and are omitted in Zilly only for the sake of minimalism, but they do make a language more human-friendly.
- Zilly employs strict evaluation: arguments are evaluated before substitution.

- *If* expressions are a special form, as is the case for conditional expressions in all strict evaluation languages. Only the predicate e_1 is eagerly evaluated; if the result is zero, the value of the *If* expression is the result of evaluating e_3 , otherwise is the result of evaluating e_2 .
- The *Defer* constructor stops the evaluation process leaving behind the enclosed expression intact. That is equivalent to saying that the r-value meta-function eliminates the *Defer* constructor at the ‘price’ of stopping right there. We often think of r-value and *Defer* as matter and anti-matter: they annihilate each other.
- For integer expressions x and y , $Less(x, y)$ and $Minus(x, y)$ correspond to the operations $x < y$ and $x - y$. $Less$ returns 1 if $x < y$ and 0 otherwise. There is no Boolean type in Zilly: the I in red stands for the indicator function 1_I , which maps true to 1 and false to 0.
- *Random* applied to an integer i returns 0 if $i < 1$, otherwise returns a random integer in the range $[0 .. i - 1]$.
- Finally, *Formula* exposes the c-value meta-function to the programmer. We could have written $Formula(x)$ instead of $Formula(e)$ in the rule, to suggest that *Formula* only applies to symbols, because that is in fact true of Zilly. Lilly, on the other hand, has container types, like arrays, where elements have their own independent l-values. We wish to emphasize that $Formula(e)$ applies in principle to any expression that has l-value.

5.3 Natural Semantics of Zilly Statements

$$\frac{\Gamma \vdash e \Downarrow e_r \quad e_r :: t}{\Gamma \vdash Define(x : t, e) \rightarrow_t \Gamma, x : (t, e_r)}$$

$$\frac{\Gamma \vdash x : t \quad e \Downarrow e_r \quad e_r :: t}{\Gamma \vdash Assign(x, e) \rightarrow_t \Gamma' = \Gamma[x : (t, e_r)/x : (t, *)]}$$

5.4 Consistency and the compute-cycle semantics

TBW

5.5 Language extensions under considerations

(optional: if space allows) TBW: lazy-ness propagation & alternating XV and RV transforms

5.6 From Zilly to Lilly: what's next?

TBW: plan to extend this work to handle containers and gradual typing

6 Related work

TBW: no pain no gain

7 Conclusions and further Work

7.1 Conclusions

The generalization works and does replicate spreadsheet behavior

7.2 Further work

To-do list:

- sum types
- other pure Lilly extensions
- extend the type system to account for effects
- parametric polymorphism ... System F level: HARD!
- last but most important WASM!! (explain why it's a priority)

8 References

8.1 the quick and dirty way - Proper BibTeX will come later

[Wozniak] Williams, Gregg; Moore, Rob (January 1985). "The Apple Story / Part 2: More History"
https://archive.org/details/BYTE_Vol_10-01_1985-01_Through_The_Hourglass/page/166/mode/2up

[BBC] <https://www.bbc.com/news/business-47802280>

[Bricklin] <http://bricklin.com/>

[AppleInsider] <https://appleinsider.com/articles/20/04/18/how-apple-owes-everything-to-its-1>

[PC Magazine] <https://books.google.com/books?id=9gIPuja3CoEC&pg=PA62#v=onepage&q&f=false>

[Browne] <https://web.archive.org/web/20170610201315/http://www.cs.umd.edu/class/spring2002/c0101/MUIseum/applications/spreadsheets/spreadsheets2.html>

[D.J. Power] <https://web.archive.org/web/20170614190135/http://dssresources.com/history/ssh>

[LIVE 2023] <https://2023.splashcon.org/home/live-2023#Call-for-Submissions>