On the Semantics for Spreadsheets with Sheet Defined Functions

Bock, Alexander Asp; Bøgholm, Thomas; Sestoft, Peter; Thomsen, Bent; Thomsen, Lone Leth

Published in:
Journal of Computer Languages

DOI (link to publication from Publisher):
10.1016/j.cola.2020.100960

Creative Commons License
CC BY-NC-ND 4.0

Publication date:
2020

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Bock, A. A., Bøgholm, T., Sestoft, P., Thomsen, B., & Thomsen, L. L. (2020). On the Semantics for Spreadsheets with Sheet Defined Functions. *Journal of Computer Languages*, *57*, Article 100960. https://doi.org/10.1016/j.cola.2020.100960

Journal of Computer Languages 57 (2020) 100960

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.editorialmanager.com/cola/default.aspx

On the semantics for spreadsheets with sheet-defined functions

Alexander Asp Bock[*,1,a], Thomas Bøgholm[1,3,b], Peter Sestoft[1,2,a], Bent Thomsen[b], Lone Leth Thomsen[b]

[a] *IT University of Copenhagen, Rued Langgaards Vej 7, Copenhagen S 2300, Denmark*
[b] *Aalborg University, Selma Lagerlöfs Vej 300, Aalborg Ø 9220, Denmark*

ARTICLE INFO

Keywords:
Spreadsheet
Semantics
Funcalc
Sheet-defined function
Recalculation

ABSTRACT

We give an operational semantics for the evaluation of spreadsheets, including sheet-defined and built-in numeric functions in the Funcalc spreadsheet platform. The semantics allows for different implementations and we discuss sheet-defined functions implemented using both interpretation and run-time code generation. The semantics specifies the expected result of a computation, also considering non-deterministic functions, independently of an evaluation mechanism. It can be extended to include the cost of formula evaluation for a cost analysis e.g. for use in parallelization of computations. An interesting future direction is to investigate experimentally how close our semantics is to that of major spreadsheet implementations.

1. Introduction

Every day spreadsheets are used by millions of people, ranging from pupils doing their school hand-ins to complex financial, medical or scientific computations. In 2017 it was estimated that there were 13–25 million spreadsheet developers worldwide [1], i.e. people developing complex computations using spreadsheets. Yet, despite their widespread use the semantics of spreadsheet computations is rather underdeveloped and it is almost impossible to analyze the computational cost of spreadsheet computations. This paper takes its outset in the semantics for simple spreadsheets sketched in section 1.8 of the book Spreadsheet Implementation Technology [2].

Inspired by Peyton Jones et al. [3] Sestoft introduced the notion of sheet-defined functions in the Funcalc spreadsheet platform [2]. Sheet-defined functions are user-defined functions that can be defined directly in the cells of special *function* sheets using the same, familiar formula syntax already known by end-users, thus sheet-defined functions bring a natural abstraction mechanism to the world of spreadsheets. Funcalc also supports the notion of array formula as found in popular spreadsheet implementations such as Excel and OpenOffice Calc. Sheet-defined functions in the Funcalc spreadsheet platform can be higher order functions, which together with first class array formulas and a few simple built-in functions such as map, reduce and fold, give a simple, yet powerful way of expressing many functions that are currently hardwired into spreadsheet platforms or have to be provided through foreign function interfaces.

For example the built-in SUMPRODUCT function from Excel, which takes as arguments two arrays, multiplies corresponding components in the given arrays, and returns the sum of those products, can be expressed as a sheet-defined function `sumproduct` where cell B1 contains the definition

= DEFINE("sumproduct", $B4$, $B2$, $B3$)

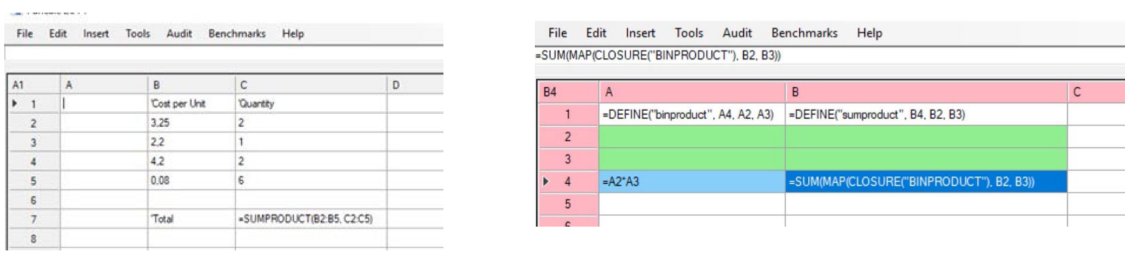
Cells B2 and B3 are input cells and cell B4 is the output cell containing the formula `=SUM(MAP(CLOSURE("BINPRODUCT"),B2,B3))`. `BINPRODUCT` is a curried version of the product operator `*` easily defined as

= DEFINE("BINPRODUCT", $A4$, $A2$, $A3$)

in cell A1 and formula `=A2*A3` in cell A4. The `sumproduct` function

* Corresponding author.
E-mail addresses: albo@itu.dk (A.A. Bock), boegholm@cs.aau.dk (T. Bøgholm), sestoft@itu.dk (P. Sestoft), bt@cs.aau.dk (B. Thomsen), lone@cs.aau.dk (L.L. Thomsen).
[1] Supported by the Independent Research Fund Denmark (grant number DFF-FTP-4005-00141), Popular Parallel Programming (P3) 2015–2019.
[2] Supported by the Sino-Danish University Center 2015–2018, PhD scholarship for Declarative Parallel Programming and the Innovation Fund Denmark (grant number 7076-00029B), Projection of Balances and Benefits in Life Insurance (ProBaBLI) 2018–2022.
[3] Supported by IT-vest (grant number AAU-2018-59).

https://doi.org/10.1016/j.cola.2020.100960
Received 11 November 2019; Received in revised form 18 January 2020; Accepted 23 February 2020
Available online 07 March 2020



A1	A	B	C	D
1		'Cost per Unit	'Quantity	
2		3.25	2	
3		2.2	1	
4		4.2	2	
5		0.08	6	
6				
7		'Total	=SUMPRODUCT(B2:B5, C2:C5)	
8				

=SUM(MAP(CLOSURE("BINPRODUCT"), B2, B3))

B4	A	B	C
1	=DEFINE("binproduct", A4, A2, A3)	=DEFINE("sumproduct", B4, B2, B3)	
2			
3			
4	=A2*A3	=SUM(MAP(CLOSURE("BINPRODUCT"), B2, B3))	
5			

Fig. 1. Definition and usage of the `sumproduct` sheet-defined function.

$$
\begin{array}{lcll}
e & ::= & n & \text{IEEE floating}-\text{point constant} \\
 & | & ca & \text{cell reference} \\
 & | & \texttt{IF}(e_1,e_2,e_3) & \text{conditional expression} \\
 & | & \texttt{RAND}() & \text{volatile function} \\
 & | & \texttt{F}(e_1,\ldots,e_n) & \text{built}-\text{in function call}
\end{array}
$$

Fig. 2. Syntax of the simplified formula language.

and its usage is shown in Fig. 1. Also note that the built-in function SUM could be defined from a curried version of the plus operator + and the REDUCE function. Including SUM, Funcalc has a number of built-in functions. In this paper, we only consider the subset of numeric built-in functions. We briefly discuss the extension of the semantics to other types of values such as character strings and dates.

As mentioned, sheet-defined functions in Funcalc are inspired by Peyton-Jones et al. [3]. Other related work include [4], where a scheme-based spreadsheet implementation is generalised, and [5], where a Haskell interpreter can be called from Excel. In a recent paper, [6] McCutchen et al. elaborate the idea of elastic sheet-defined functions that generalises spreadsheet functions to variable sized input arrays. The rationale for the design of elastic sheet-defined functions is that it avoides the use of map, reduce, fold, and other aggregate operations from classic functional programming. However, elastic sheet-defined functions require a subtle generalization algorithm. [6] presents a correctness proof for the algorithm based on a semantics for spreadsheets with elastic sheet-defined functions. The semantics is given in terms of a set of recursive equations, which amount to a denotational semantics of formulas.

In this paper we give a simple but precise operational semantics for the evaluation of extended spreadsheet formulas, with array formulas, sheet-defined functions and closures, as found in the Funcalc spreadsheet platform [2].

The purpose of this semantics is to serve as a guideline for implementation work, and we discuss how the semantics has guided two different implementations of sheet-defined functions. The semantics is also a starting point for further work on defining cost semantics for spreadsheet calculations which may be useful for guiding parallelisation of spreadsheet computations.

The rest of this paper is organized as follows: The evaluation semantics for simple Funcalc expressions is elaborated in Section 2 and semantics for extended spreadsheet expressions is developed in Section 3. We discuss how the implementation of sheet-defined functions respects important aspects of the semantics in Section 4. Finally, we present conclusions and future work in Section 5.

2. Simple spreadsheet semantics

This section describes the evaluation of simple spreadsheets without array formulas and sheet-defined functions, and hence without array values and closures. It is reproduced from parts of [2, Section 1.8] and included here for background; readers familiar with the subject may skip to Section 3.

The simplified formulas used in this section are described in Fig. 2. One simplification is to represent a constant cell `n` by a constant formula `=n`, although most spreadsheet programs would distinguish them. Another simplification is to leave out cell area expressions ca_1: ca_2; these will be introduced in Section 3.

To describe the evaluation of such formulas, we use the semantic sets and functions defined in Fig. 3. These are sometimes called semantic domains, but here they are ordinary sets and partial functions. For instance, *Value* = *Number* + *Error* is the set of values, where a value v is either a (finite, non-NaN) IEEE 854 binary floating-point number such as 0.42 in set *Number* or an error such as `#DIV/0!` in set *Error*. The set *Addr* contains cell addresses *ca* such as B2. For presentational simplicity, some additional error values (such as `#NAME!`) and additional kinds of values (such as strings), found in realistic spreadsheet programs, have been left out. They are easily added to the semantics studied here.

$$
\begin{array}{llll}
n & \in & \mathit{Number} & = \{\ \text{IEEE floating}-\text{point numbers}\ \} \\
 & & \mathit{Error} & = \{\ \texttt{\#DIV/0!},\ \texttt{\#CYCLE!}\ \} \\
ca & \in & \mathit{Addr} & = \{\ \text{cell addresses}\ (c, r)\} \\
v & \in & \mathit{Value} & = \mathit{Number} + \mathit{Error} \\
e & \in & \mathit{Expr} & = \{\ \text{formulas, see Figure 2}\ \} \\
\phi & & & \in \mathit{Addr} \to \mathit{Expr} \\
\sigma & & & \in \mathit{Addr} \to \mathit{Value}
\end{array}
$$

Fig. 3. Sets and maps used in the spreadsheet semantics: *Number* is the set of proper IEEE 854 binary floating-point numbers, excluding NaNs and infinities; *Error* is the set of error values; *Addr* the set of cell addresses, each a pair (*c, r*) of column and row number; *Value* the set of values (either number or error); and *Expr* the set of formulas.

To describe the formulas of a worksheet, we use a map ϕ: *Addr* → *Expr* so that when $ca \in Addr$ is a cell address, $\phi(ca)$ is the formula in cell *ca*. If cell *ca* is blank, then $\phi(ca)$ is undefined. The domain of ϕ is $dom(\phi) = \{\ ca \mid \phi(ca) \text{ is defined}\ \}$, the set of cell addresses that have a formula, that is, the set of non-blank cells. The ϕ function is not affected by recalculation, only by editing the sheet.

The result of a recalculation is modelled by function σ: *Addr* → *Value*, where $\sigma(ca)$ is the computed value in cell *ca*. The σ function gets updated by each recalculation (see Section 2.2).

2.1. Semantics of formula evaluation

The semantics for formulas is given as a natural semantics [7], a variant of operational semantics [8], using inference rules that involve big-step evaluation judgments. An evaluation judgment has the form $\sigma \vdash e \Downarrow v$, which says: When σ describes the calculated values of all cells, then formula e may evaluate to value v. Note that v may be a number value or an error value.

To understand inference rules, consider this rule:

$$\frac{\sigma \vdash e_i \Downarrow v_i \in Error}{\sigma \vdash \mathrm{F}(e_1, \ldots, e_n) \Downarrow v_i} (e5e)$$

This inference rule consists of a premise above the line and a conclusion below the line. The conclusion concerns the value of a function call expression $\mathrm{F}(e_1, \ldots, e_n)$, and the premise concerns the value of one of the call's argument expressions e_i. The rule can be read as follows: If there is some argument expression e_i that may evaluate to an error value v_i, then the function call may evaluate to the error value v_i also. That is, the rule describes the propagation of errors from argument to result in a function call. If multiple arguments e_i and e_j may evaluate to different error values v_i and v_j, then the rule does not specify which error will be propagated to the call's result.

For another example, consider this rule, also for a function call $\mathrm{F}(e_1, \ldots, e_n)$ with n arguments:

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \notin Error \ldots \sigma \vdash e_n \Downarrow v_n \notin Error}{\sigma \vdash \mathrm{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)} (e5v)$$

This rule has n premises and can be read as follows: If all argument expressions $e_1, \ldots, e_n$ may evaluate to non-error values $v_1, \ldots, v_n$, then the value of the function call is obtained by applying the actual function f to these values, as in $f(v_1, \ldots, v_n)$.

The "may" is important because, in general, an expression may evaluate to multiple different values. For instance, `RAND()` may evaluate to any number between 0.0 (included) and 1.0 (excluded). Hence, `7+1/RAND()` may evaluate to some number greater than 7 + 1 or to the error `#DIV/0!` in case `RAND()` produces 0.0.

The complete set of inference rules that describe when a formula evaluation judgment $\sigma \vdash e \Downarrow v$ holds is given in Fig. 4. Note that there are five groups of rules ($e1$), ($e2x$), ($e3x$), (4), ($e5x$), each corresponding to one of the five kinds of formulas in Fig. 2. Also, the formula fragments that appear in the premises are always smaller than the formula that appears in the conclusion. Hence, one can make a conclusion about a given formula through a finite number of rule applications.

The formula evaluation rules in Fig. 4 may be explained as follows:

Rule ($e1$) says that a number constant `n` evaluates to that constant's value.

Rule ($e2b$) says that a reference ca to a blank cell, that is, one for which $\sigma(ca)$ is not defined, gives value 0.0.

Rule ($e2v$) says that a reference ca to a non-blank cell evaluates to the value $\sigma(ca)$ calculated for that cell. This value may be a number or an error.

Rule ($e3e$) says that the expression $\mathrm{IF}(e_1, e_2, e_3)$ may evaluate to error v_1 if the condition e_1 may evaluate to error v_1.

Rule ($e3f$) says that $\mathrm{IF}(e_1, e_2, e_3)$ may evaluate to value v provided the condition e_1 may evaluate to the non-error number zero and the "false branch" e_3 may evaluate to v.

Rule ($e3t$) says that $\mathrm{IF}(e_1, e_2, e_3)$ may evaluate to value v provided the condition e_1 may evaluate to some non-error non-zero number v_1 and the "true branch" e_2 may evaluate to v. Since Funcalc does not have a boolean type, floating-point numbers are used instead, where 0.0 is considered false and any number different from 0.0 is considered true. Note that although in numeric software it is bad practice to compare floating-point numbers for equality, an IEEE floating-point number either is or is not equal to zero, so semantically the comparison $v_1 \neq 0.0$ is unproblematic.

Rule ($e4$) says that function call RAND() may evaluate to any (non-error) number v greater than or equal to zero and less than one. Hence, this rule models nondeterministic choice. It permits a formula involving `RAND()` to produce a different result on each evaluation. However, it does not *require* `RAND()` to produce a different number every time it is called. Such a requirement would not make sense; by definition, a random number generator is permitted to return whatever result it wants. So according to this operational semantics, `RAND()` might consistently return 0.42 whenever it is called, although that would be rather disappointing and useless.

Rule ($e5e$) says that a call $\mathrm{F}(e_1, \ldots, e_n)$ to a built-in function `F` may evaluate to error v_i if one of its arguments e_i may evaluate to error v_i. Note that if more than one argument may evaluate to an error, then the function call may evaluate to any of these. Hence, the semantics does not prescribe an evaluation order for arguments, such as a left to right or right to left.

Rule ($e5v$) says that a call $\mathrm{F}(e_1, \ldots, e_n)$ to a function `F` may evaluate to value v if each argument e_i may evaluate to non-error value v_i, and applying the actual function f to arguments $(v_1, \ldots, v_n)$ produces value v. The final result v may be a number such as 5, for instance, if the call is `+(2, 3)`; or it may be an error such as `#DIV/0!`, for instance, if the call is `/(1.0, 0.0)`.

2.2. Semantics of simple recalculation

Now that we know how to evaluate a formula, given values of all cells in the worksheet, we can describe the semantics of a recalculation. A recalculation must find a value for every non-blank cell ca in the sheet, and that value $\sigma(ca)$ must agree with the formula $\phi(ca)$ held in that cell. These are the central consistency requirements on a recalculation, formally described in Fig. 5. These requirements leave it completely unspecified how the recalculation works, whether it recalculates all or only some cells, whether it does so sequentially or in parallel, whether it guesses the values or computes them, and so on. This underspecification is essential to permit a range of implementation strategies and optimizations.

3. Funcalc semantics

In this section, we extend the simple spreadsheet semantics from Section 2. We first extend the expressions and semantic sets to account for array formulas and sheet-defined functions. We then discuss the modelling of array formulas, which is slightly more general than strictly necessary. With array formulas in the expression language, we need to extend the semantics for ordinary data sheets. This turns out to be a smooth extension where "old" rules just pass around an additional semantic environment for array expressions. We then extend the semantics to account for function sheets, special sheets where one can define sheet-defined functions directly in the spreadsheet paradigm, and round of the section with a discussion of the rules for calling sheet-defined functions, as these rules are some of the more unusual aspects of this semantics.

3.1. Extended expressions and semantic sets

The simple spreadsheet semantics from Section 2 must be expanded in two orthogonal directions: to account for array formulas and to account for sheet-defined functions. This requires extension to the formula expression language, shown in Fig. 6, and to the set of values and semantic maps, shown in Fig. 7.

A cell area reference ca_1: ca_2 refers to a block of cells spanned by the two opposing "corner" cells ca_1 and ca_2. In Funcalc, a cell area reference can refer to an ordinary sheet only, not to a function sheet.

An array formula is here modelled as an underlying formula ae which is itself just an expression, expected to evaluate to an array value, that is, an array of values. That array value's components are distributed over a target cell area, with one such component in each cell. This is explained in more detail in Section 3.2.

We model a closure as a partial application, that is, a named sheet-defined function sdf with a prefix $[u_1, \ldots, u_k]$ of its argument values given, where $0 \leq k \leq arity(sdf)$; see Fig. 7. A closure is created by `CLOSURE` from a sheet-defined function sdf by giving it values for some or all of its arguments. A partially applied closure e_0 may be given further arguments, as in currying, also using `CLOSURE`. An `APPLY` call of a closure e_0 must provide all the remaining n arguments, where

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma \vdash ca \Downarrow v}\ (e2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \in Error}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v_1}\ (e3e)$$

$$\frac{\sigma \vdash e_1 \Downarrow 0.0 \qquad \sigma \vdash e_3 \Downarrow v}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v}\ (e3f)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad v_1 \neq 0.0 \qquad \sigma \vdash e_2 \Downarrow v}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v}\ (e3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma \vdash \mathtt{RAND}() \Downarrow v}\ (e4)$$

$$\frac{\sigma \vdash e_i \Downarrow v_i \in Error}{\sigma \vdash \mathtt{F}(e_1,\ldots,e_n) \Downarrow v_i}\ (e5e)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \notin Error \qquad \ldots \qquad \sigma \vdash e_n \Downarrow v_n \notin Error}{\sigma \vdash \mathtt{F}(e_1,\ldots,e_n) \Downarrow f(v_1,\ldots,v_n)}\ (e5v)$$

Fig. 4. Evaluation rules for simplified spreadsheet formulas.

$$\begin{array}{ll} (1) & dom(\sigma) = dom(\phi) \\ (2) & \forall ca \in dom(\phi).\ \sigma \vdash \phi(ca) \Downarrow \sigma(ca) \end{array}$$

Fig. 5. The consistency requirements on recalculation for simple formulas. Requirement (1) says that a recalculation must find a value $\sigma(ca)$, possibly an error, for every non-blank cell *ca*. Requirement (2) says that the computed value $\sigma(ca)$ must agree with the cell's formula $\phi(ca)$.

$k + n = arity(sdf)$, and will call the underlying sheet-defined function.

While the extended semantics has a richer set of values, as shown in Fig. 7, we still only consider two kinds of base values: floating-point numbers and error values. We could additionally consider other base values, such as character strings. However, in some spreadsheet implementations, these have highly idiosyncratic implicit conversions. For instance, in Excel, `"abc"+45` evaluates to an error, whereas `"123"+45` evaluates to the number 168. So does `SUM("123",45)`, but `SUM(A1:A2)` evaluates to 45 when cell A1 contains `"123"` and A2 contains `45`. So in numeric addition, a character string either produces an error, or is converted to a number, or is ignored. Whereas general spreadsheet behaviour seems rational and elegant, this appears to call for rather ad hoc and inelegant rules. For this reason, and because character strings and other base values pose no new problems and their inclusion in the semantics provides no new insights, we ignore them here. Moreover, note that in most spreadsheet implementations, a truth value such as `FALSE`, or a date such as `2019-12-29`, are just fancily formatted numbers.

In the simple semantics for formula evaluation on ordinary sheets, the recalculation consistency requirements could be stated in terms of the formula $\phi(ca)$ in a given cell and its post-recalculation value $\sigma(ca)$.

To account for array formulas, we need the post-recalculation value $\alpha(ae)$ of each underlying (presumably array-valued) expression *ae*. This underlying value will be shared by all the array formula's components, see rule (*e*7) in Fig. 11. In Funcalc, array formulas are allowed on ordinary sheets only, not on function sheets.

To further account for a call to a sheet-defined function, we need the value $\rho(ca)$ of the function sheet cells *ca* used during the call of the function. Each call, also each recursive call, has its own fresh ρ map, and the map is ephemeral: there is no way to refer to a function sheet cell value after the function has returned. Hence ρ is similar to a stack frame in ordinary programming language implementation.

Note that *α* is not needed when evaluating a sheet-defined function, because function sheets cannot contain array formulas. Also, ρ is not needed when evaluating cells on an ordinary sheet, because there is no way to refer to a function sheet cell value after the function has returned. The revised post-recalculation consistency requirements are shown in Fig. 8.

3.2. Modelling array formulas

An array formula is an expression, such as `transpose(e2:g3)`, whose result is an array value and where the components of this array value are spread over a target cell area, such as B2:C4. This situation is shown in Fig. 9 where the target cell area has been marked and the formula has been written into cell B2; the array formula is then usually created by pressing a key combination such as Ctrl + Shift + Enter in Excel. The effect of doing so is shown in Fig. 10 where the target cells B2:C4 contain the transpose of the values in E2:G3.

Editing any cell in the range E2:G3 would cause the array expression to be recalculated and the values in B2:C4 to be updated. The underlying array expression is evaluated at most once in each recalculation.

In the extended spreadsheet expressions of Funcalc, we model the individual cells belonging to an array formula by the syntax *ae*[*i, j*] that suggests indexing into the value of the underlying array expression *ae*. In the index (*i, j*) the *i* and *j* are constants, with *i* ranging over columns and *j* over rows, both one-based. For instance, cell B2 in Figs. 9 and 10

$$\begin{array}{llll}
e & ::= & n & \text{number constant} \\
 & | & ca & \text{cell reference} \\
 & | & \mathtt{IF}(e_1,e_2,e_3) & \text{conditional expression} \\
 & | & \mathtt{RAND}() & \text{volatile function} \\
 & | & \mathtt{F}(e_1,\ldots,e_n) & \text{call to built}-\text{in function} \\
 & | & ca_1 : ca_2 & \text{cell area reference} \\
 & | & ae[i,j] & \text{array formula component} \\
 & | & sdf(e_1,\ldots,e_n) & \text{call to sheet}-\text{defined function} \\
 & | & \mathtt{CLOSURE}(sdf,e_1,\ldots,e_k) & \text{closure creation} \\
 & | & \mathtt{CLOSURE}(e_0,e_1,\ldots,e_n) & \text{closure partial application} \\
 & | & \mathtt{APPLY}(e_0,e_1,\ldots,e_n) & \text{closure full application} \\
ae & ::= & e & \text{array expression}
\end{array}$$

Fig. 6. Syntax of the Funcalc extended formula language, with five additional syntactic constructs: a cell area reference, an access to component (*i, j*) of an array formula *ae*, a call of a sheet-defined function, creation of a closure from a sheet-defined function *sdf*, and call of a closure e_0.

$$\begin{array}{llll}
n & \in & Number & = \{\ \text{IEEE floating}-\text{point numbers}\ \} \\
av & \in & ArrVal & = \{\ (w, h, [[v_{ij} \mid i \le w, j \le h]])\ \} \\
fv & \in & FunVal & = \{\ (sdf, [u_1, \ldots, u_k])\ \} \\
 & & Error & = \{\ \texttt{\#DIV/0!},\ \texttt{\#CYCLE!}\ \} \\
ca & \in & Addr & = \{\ \text{cell addresses}\ (c, r) \} \\
v, u & \in & Value & = Number + Error + ArrVal + FunVal \\
e & \in & Expr & = \{\ \text{formulas, see Figure 6}\ \} \\
\phi & & & \in Addr \to Expr \\
\sigma & & & \in Addr \to Value \\
\alpha & & & \in Expr \to Value \\
\rho & & & \in Addr \to Value
\end{array}$$

Fig. 7. Sets and maps used in the Funcalc extended spreadsheet semantics. There are the following differences relative to Fig. 3: $av \in ArrVal$ is an array value with $w \times h$ component values v_{ij} and $fv \in FunVal$ is a function value (closure) consisting of a function name sdf and $0 \le k \le arity(sdf)$ given argument values u_i. In this extended semantics, $v \in Value$ is either a number or error or array value or function value. Array values are needed because of cell area expressions ca_1: ca_2, and function values because of `CLOSURE` expressions. There are new semantic maps: α maps an array expression ae to its value, and ρ maps a function sheet cell address to its value.

$$\begin{array}{ll}
(1) & dom(\sigma) = dom(\phi) \\
(2) & \forall ca \in dom(\phi).\ \sigma, \alpha \vdash \phi(ca) \Downarrow \sigma(ca) \\
(3) & \forall ae \in dom(\alpha).\ \sigma, \alpha \vdash ae \Downarrow \alpha(ae)
\end{array}$$

Fig. 8. The consistency requirements on recalculation with array formulas and sheet-defined functions. The requirement (2) is extended with α to account for array formulas. The new requirement (3) says that a recalculation must find a single value $\alpha(ae)$ for each array expression ae underlying an array formula; this value will be used in all components of the array formula via applications of (2).

would contain the expression $ae[1, 1]$ where ae is the underlying array expression `transpose(e2:g3)`, cell B3 would contain $ae[1, 2]$, cell C2 would contain $ae[2, 1]$, and so on. Indexing into an error value produces that error value itself, so we need no separate "error version" of rule ($e7$) in Fig. 11.

The syntax in Fig. 6 allows an array formula component $ae[i, j]$ to appear anywhere an expression can, also nested inside another expression. This is overly general, since $ae[i, j]$ need appear only at top level in a cell formula, not in nested expressions. We could enforce this restriction by introducing an additional syntactic category of *cell* which can be an expression e or an array formula component $ae[i, j]$, and remove the latter from the syntactic category of expressions. This would also be in better agreement with the implementation of Funcalc. However, since the excess generality is harmless, and getting rid of it would lead to additional largely administrative rules without semantic significance, we stick to the simple but slightly too permissive syntax in Fig. 6.

3.3. Extended semantics for ordinary sheets

An evaluation judgment in the extended semantics for ordinary formula evaluation has the form $\sigma, \alpha \vdash e \Downarrow v$. It says that when σ describes the calculated values of all cells and α describes the values of all array expressions underlying array formulas, then formula e may evaluate to value v, where v may be a number value, an error value, an array value or a closure value.

The rules defining the judgment $\sigma, \alpha \vdash e \Downarrow v$ are shown in Fig. 11. They are a smooth extension of those in Fig. 4. The "old" rules ($e1$) through ($e5v$) have been extended to also pass around the α array expression map. The six new rules ($e6$), ($e7$), ($e8$), ($e9$), ($e10$), and ($e11$) account for cell area references, array formulas, calls to sheet-defined functions, closure creation, and closure calls. They correspond exactly to the six new syntactic constructs in Fig. 6.

In Fig. 11, the new rule ($e6$) says that a cell area reference ca_1: ca_2 evaluates to an array value $ArrVal(w, h, [[v_{ij}]])$ with w columns, h rows, and $w \cdot h$ values v_{ij} obtained from the cell value map σ. Here, we assume we can look up values in σ using indices. The utility function $sort(x, y)$ returns the pair of the least and greatest of x and y, so that c_l and c_r are the leftmost and rightmost column indexes, and r_t and r_b are the top and bottom row indexes, of the cell area ca_1: ca_2. This is necessary because it is legal to enter a cell area reference such as B1:A2, thereby giving the cell area's upper right (B1) and lower left (A2) corners, that should be "normalized" to A1:B2 which gives the cell area's upper left and lower right corners instead, for proper calculation of width and height.

Rule ($e7$) says that component (i, j) of an array formula is found by looking up the value $av = \alpha(ae)$ of the underlying array expression and then indexing into that value by $av[i, j]$, where such indexing must produce an error value if av is not an array value or does not have a component (i, j). Note that in the judgment left-hand side, the indexing notation is syntactic, and in the right-hand side it is semantic.

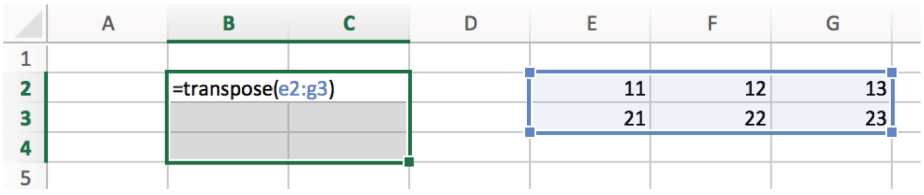
Rule ($e8$) describes how to call a sheet-defined function sdf that has output cell address out, that has input cell addresses $[in_1, \ldots, in_n]$, and that is defined using only cells at addresses $cells$ (excluding the input cells but including the output cell) on a separate function sheet. We assume that the cells defining sdf are given by $def(sdf) = (out, [in_1, \ldots, in_{k+n}], cells)$, and that all these cells are in $dom(\phi)$ — that is, ϕ describes also the formulas of the function sheet on which sdf is defined.

The evaluation of a call to a sheet-defined function sdf proceeds as follows. First evaluate the call's argument expressions to values $v_1, \ldots, v_n$, then postulate a fresh environment ρ' in which the called function's input cells $[in_1, \ldots, in_n]$ have these values and all other cells used by the function have consistent values. Then the function call's value is the value $\rho'(out)$ of the function's output cell. Judgments of the form ρ', $\sigma \vdash e \Downarrow v$ are defined in Fig. 12 below. For a discussion of the function call semantics, and an example, see Section 3.5.1.

It is natural to expect the new ρ' environment to be defined for all the sheet-defined function's cells, as in $dom(\rho') = \{in_1, \ldots, in_n\} \cup cells$, but actually it suffices for $dom(\rho')$ to be the set of cells needed to compute the value of the output cell out.

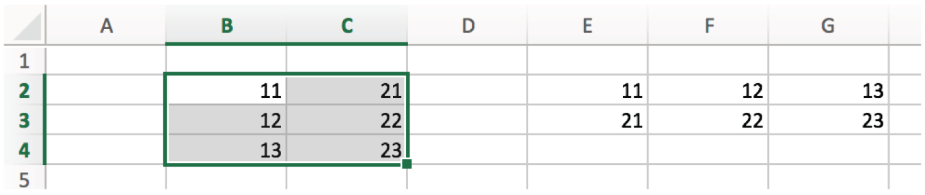
The expression language is call-by-value, and a call to a sheet-defined function is strict in the sense that every argument is evaluated before the function is called, regardless of whether the function's body actually refers to the argument's value.

A call to a sheet-defined function is not error-strict: the function is called even though some argument e_i evaluates to an error value. Hence



	A	B	C	D	E	F	G
1							
2		=transpose(e2:g3)			11	12	13
3					21	22	23
4							
5							

Fig. 9. Entering an array formula with target cell area B2:C4.



	A	B	C	D	E	F	G
1							
2		11	21		11	12	13
3		12	22		21	22	23
4		13	23				
5							

Fig. 10. The six cells in target cell area B2:C4 each contain one component of the result of the underlying array expression `transpose(e2:g3)`.

the argument evaluation premises are simpler than in rule (*e*5*v*) for calling built-in functions, and there is no error-case rule (*e*8*e*) corresponding to rule (*e*5*e*). Naturally, the same holds for constructing or calling a closure in rules (*e*9), (*e*10) and (*e*11).

Rule (*e*9) says that to evaluate a closure creation expression, evaluate the *k* given arguments and create a function value consisting of the function name *sdf* and the *k* resulting values, whether errors or proper values.

Rule (*e*10) says that to further apply a partially applied closure, evaluate e_0 to a closure containing *k* already given arguments, then evaluate the *n* further arguments, and create a new closure containing the $k + n \leq arity(sdf)$ argument values given so far.

Rule (*e*11) says that to evaluate a call to closure $FunVal(sdf, [u_1, ..., u_k])$, evaluate the *n* given arguments, then proceed to call the sheet-defined function *sdf* on its $k + n$ arguments in the same manner as described in rule (*e*8).

3.4. Extended semantics for function sheets

An evaluation judgment in the semantics for sheet-defined functions has the form $\rho, \sigma \vdash e \Downarrow v$. It says that when ρ describes the calculated values of all cells on the function sheet defining the function and *σ* describes the calculated values of all cells on ordinary sheets, then formula *e* may evaluate to value *v*, where *v* may be a number value, an error value, an array value or a closure value.

The rules defining the judgment $\rho, \sigma \vdash e \Downarrow v$ are shown in Fig. 12. They are intentionally very similar to those for evaluation of ordinary (extended) spreadsheet formulas shown in Fig. 11 — after all, the whole point of sheet-defined functions is that they should be familiar to spreadsheet users.

However, there is an additional rule (*f*2*f*) for lookup of a cell address on a function sheet; rule (*f*6) requires a cell area reference to refer to an ordinary worksheet, not a function sheet; and there is no rule (*f*7) for array formulas, which are not allowed in function sheets.

$$\frac{}{\sigma, \alpha \vdash n \Downarrow n}\ (e1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma, \alpha \vdash ca \Downarrow 0.0}\ (e2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma, \alpha \vdash ca \Downarrow v}\ (e2v)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \in Error}{\sigma, \alpha \vdash \mathtt{IF}(e_1, e_2, e_3) \Downarrow v_1}\ (e3e)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow 0.0 \qquad \sigma, \alpha \vdash e_3 \Downarrow v}{\sigma, \alpha \vdash \mathtt{IF}(e_1, e_2, e_3) \Downarrow v}\ (e3f)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \qquad v_1 \neq 0.0 \qquad \sigma, \alpha \vdash e_2 \Downarrow v}{\sigma, \alpha \vdash \mathtt{IF}(e_1, e_2, e_3) \Downarrow v}\ (e3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma, \alpha \vdash \mathtt{RAND}() \Downarrow v}\ (e4)$$

$$\frac{\sigma, \alpha \vdash e_i \Downarrow v_i \in Error}{\sigma, \alpha \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow v_i}\ (e5e)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \notin Error \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n \notin Error}{\sigma, \alpha \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)}\ (e5v)$$

Fig. 11. Evaluation rules for Funcalc extended spreadsheet formulas.

$$\frac{(c_1,r_1)=ca_1 \qquad (c_2,r_2)=ca_2 \qquad (c_l,c_r)=sort(c_1,c_2) \qquad (r_t,r_b)=sort(r_1,r_2) \qquad w=c_r-c_l+1 \qquad h=r_b-r_t+1}{\sigma,\alpha \vdash ca_1 : ca_2 \Downarrow ArrVal(w,h,[[\sigma[c_l+i,r_t+j] \mid i\le w, j\le h]])}\ (e6)$$

$$\frac{}{\sigma,\alpha \vdash ae[i,j] \Downarrow \alpha(ae)[i,j]}\ (e7)$$

$$\frac{\sigma,\alpha\vdash e_1\Downarrow v_1 \quad \dots \quad \sigma,\alpha\vdash e_n\Downarrow v_n \qquad def(sdf)=(out,[in_1,\dots,in_n],cells) \qquad \rho'\ \text{fresh} \quad \rho'(in_1)=v_1 \quad \dots \quad \rho'(in_n)=v_n \qquad \forall ca\in dom(\rho')\setminus\{in_1,\dots,in_n\}.\ \rho',\sigma\vdash\phi(ca)\Downarrow\rho'(ca)}{\sigma,\alpha\vdash sdf(e_1,\dots,e_n)\Downarrow\rho'(out)}\ (e8)$$

$$\frac{\sigma,\alpha\vdash e_1\Downarrow u_1 \quad \dots \quad \sigma,\alpha\vdash e_k\Downarrow u_k}{\sigma,\alpha\vdash \mathtt{CLOSURE}(sdf,e_1,\dots,e_k)\Downarrow FunVal(sdf,[u_1,\dots,u_k])}\ (e9)$$

$$\frac{\sigma,\alpha\vdash e_0\Downarrow FunVal(sdf,[u_1,\dots,u_k]) \qquad \sigma,\alpha\vdash e_1\Downarrow v_1 \quad \dots \quad \sigma,\alpha\vdash e_n\Downarrow v_n}{\sigma,\alpha\vdash \mathtt{CLOSURE}(e_0,e_1,\dots,e_n)\Downarrow FunVal(sdf,[u_1,\dots,u_k,v_1,\dots,v_n])}\ (e10)$$

$$\frac{\sigma,\alpha\vdash e_0\Downarrow FunVal(sdf,[u_1,\dots,u_k]) \qquad \sigma,\alpha\vdash e_1\Downarrow v_1 \quad \dots \quad \sigma,\alpha\vdash e_n\Downarrow v_n \qquad def(sdf)=(out,[in_1,\dots,in_{k+n}],cells) \qquad \rho'\ \text{fresh} \quad \rho'(in_1)=u_1\ \dots\ \rho'(in_k)=u_k \qquad \rho'(in_{k+1})=v_1\ \dots\ \rho'(in_{k+n})=v_n \qquad \forall ca\in dom(\rho')\setminus\{in_1,\dots,in_{k+n}\}.\ \rho',\sigma\vdash\phi(ca)\Downarrow\rho'(ca)}{\sigma,\alpha\vdash \mathtt{APPLY}(e_0,e_1,\dots,e_n)\Downarrow\rho'(out)}\ (e11)$$

Fig. 11. (*continued*)

3.5. Discussion

3.5.1. Calling a sheet-defined function

The rules ($e8$) and ($f8$) for calls to sheet-defined functions, and the corresponding closure call rules ($e11$) and ($f11$), are some of the more unusual aspects of this semantics. The core idea in these rules is that a fresh environment ρ′ is postulated for evaluation of the called function *sdf*. Informally, this corresponds to (A) the creation of a fresh copy of the function sheet on which *sdf* is defined, and also to (B) the creation of a new stack frame to hold the function's arguments and local variables. Explanation (A) is what a Funcalc spreadsheet user should have in mind, and (B) is what the Funcalc implementation actually does. Without loss of generality we can assume that each sheet-defined function is defined in its own sheet, and only the cells used in the definition of *sdf* need be recalculated.

There is nothing mysterious or unusual about the "freshness" of ρ′. Formally, ρ′ is no different from v in rule ($e4$): it is just a variable representing some value (here an environment) that must satisfy the premises.

In explanation (A), the fresh sheet copy ρ′ is used as follows: fill the input cells $[in_1, ..., in_n]$ with the values of the evaluated arguments; recalculate the sheet as usual for spreadsheets; return the output cell's value as the result of the call; and discard the sheet copy. These steps are faithfully reflected in rules ($e8$) and ($f8$), with the "recalculate as usual" step expressed by the last premise

$$\forall\ ca \in dom(\rho')\setminus\{in_1,...,in_n\}.\ \ \rho',\ \sigma\vdash\phi(ca)\Downarrow\rho'(ca)$$

This premise is meant to reflect the standard spreadsheet consistency requirement (2) in Figs. 5 and 8 but for the temporary function sheet's cell values in ρ′ instead of an ordinary sheet's cell values in σ.

Consider the simple sheet-defined function `F` in Fig. 13, with input cells B2 and B3, intermediate cell B4 containing the formula `=IF(RAND()<0.5, B2, B3)` and output cell B5 containing the formula `=B4+B4`.

The set *cells* of cells making up the function's body is {B4, B5}. To evaluate a call `F(1,5)` to this function, the semantics will create a fresh environment ρ′ that must have ρ'(B2) = 1 and ρ'(B3) = 5. Now we can additionally have either ρ'(B4) = 1 and so ρ'(B5) = 2, or ρ'(B4) = 5 and so ρ'(B5) = 10; these are the only two possibilities according to the semantics. Hence the call `F(1,5)` must return 2 or 10. It cannot return 1 + 5 = 6 because both occurrences of B4 in `=B4+B4` must have the same value ρ′(B4).

Note that the universal quantification ∀ $ca \in dom(\rho')$... in the last premise of rules ($e8$), ($f8$), ($e11$) and ($f11$) ranges over a finite set: the cells used to define function *sdf*. Hence in any concrete application of these rules, the quantifier gives rise to only a finite number of proof subtrees.

A call from a sheet-defined function to another one, or indeed a recursive call to the function itself, is handled naturally by rules ($f8$) and ($f11$) through postulating a new fresh environment ρ′ for the called function, distinct from the calling function's ρ. In terms of explanation

$$\frac{}{\rho, \sigma \vdash n \Downarrow n}\ (f1)$$

$$\frac{ca \notin dom(\rho) \qquad ca \notin dom(\sigma)}{\rho, \sigma \vdash ca \Downarrow 0.0}\ (f2b)$$

$$\frac{ca \in dom(\rho) \qquad \rho(ca) = v}{\rho, \sigma \vdash ca \Downarrow v}\ (f2f)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\rho, \sigma \vdash ca \Downarrow v}\ (f2v)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow v_1 \in Error}{\rho, \sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v_1}\ (f3e)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow 0.0 \qquad \rho, \sigma \vdash e_3 \Downarrow v}{\rho, \sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v}\ (f3f)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow v_1 \qquad v_1 \neq 0.0 \qquad \rho, \sigma \vdash e_2 \Downarrow v}{\rho, \sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v}\ (f3t)$$

$$\frac{0.0 \leq v < 1.0}{\rho, \sigma \vdash \texttt{RAND}() \Downarrow v}\ (f4)$$

$$\frac{\rho, \sigma \vdash e_i \Downarrow v_i \in Error}{\rho, \sigma \vdash \texttt{F}(e_1,\ldots,e_n) \Downarrow v_i}\ (f5e)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow v_1 \notin Error \qquad \ldots \qquad \rho, \sigma \vdash e_n \Downarrow v_n \notin Error}{\rho, \sigma \vdash \texttt{F}(e_1,\ldots,e_n) \Downarrow f(v_1,\ldots,v_n)}\ (f5v)$$

Fig. 12. Evaluation rules for Funcalc sheet-defined functions.

(A) given above, a fresh copy of the defining function sheet is created for each recursive call; and in terms of (B), a fresh stack frame is allocated for each recursive call. Also, all these sheet copies, or stack frames, coexist until the function calls return. (However, as a semantics-preserving optimization, the actual Funcalc implementation may deallocate the old stack frame early in case of a tail call).

Infinite recursion in a sheet-defined function is reflected in the operational semantics by an attempt to build an infinitely deep derivation tree, through an infinite number of applications of rules (*f*8) or (*f*11). Obviously this is not possible, so no value can be derived for an infinite recursive call, not even an error value. Note that this is different from the meaning of a cyclic dependency in an ordinary spreadsheet, for which an error value could be derived (by the semantics and the implementation): just put $\sigma(ca)$=`#CYCLE!` $\in$ *Error* for all the cells *ca* cyclically dependent on each other.

3.5.2. *Appropriateness of the semantics*

The formal semantics presented here attempts to specify in a precise manner what results should be expected from recalculation in a spreadsheet implementation conforming to our semantics, in the presence of volatile (non-deterministic) functions, and without assuming anything about the evaluation order or implementation mechanism: sequential, left-to-right, right-to-left, parallel, speculative, caching, and so on. Given that major spreadsheet implementations (Microsoft Excel, LibreOffice Calc, Gnumeric, Google Sheets) do not explicitly describe their expected behaviour, it is impossible to prove that our semantics is the one intended by these implementations.

Nevertheless, we believe that our semantics accurately reflects the general expectation that after a recalculation, and in the absence of cyclic dependencies, the values exhibited by all cells in a spreadsheet are consistent with each other. We have then conservatively extrapolated this expectation to cover also Funcalc's sheet-defined functions, requiring that a sheet-defined function must behave as if evaluated on a fresh copy of the sheet defining it; hence the fresh ρ' in rules (e8), (e11), (f8) and (f11) above.

It is also our belief that in most circumstances, and in the absence of cyclic dependencies, the major spreadsheet implementations agree with the semantics given here. One known exception is caused by the so-called Data Table feature in Excel, where table entries are not recalculated correctly (that is, are left inconsistent) when the "inputs" (row or column margins) of one data table depend on a result computed by another data table [2, Page 145]. Presumably this unusual scenario was never intended to work, and LibreOffice Calc explicitly forbids it.

In the presence of a cyclic dependency, all known spreadsheet implementations will stop recalculation at some point, leave the spreadsheet in a possibly inconsistent state (violating the conditions in Figs. 5

$$\frac{\begin{array}{c} ca_1 \in dom(\sigma) \qquad ca_2 \in dom(\sigma) \\ (c_1, r_1) = ca_1 \qquad (c_2, r_2) = ca_2 \qquad (c_l, c_r) = sort(c_1, c_2) \qquad (r_t, r_b) = sort(r_1, r_2) \\ w = c_r - c_l + 1 \qquad h = r_b - r_t + 1 \end{array}}{\rho, \sigma \vdash ca_1 : ca_2 \Downarrow ArrVal(w, h, [[\sigma[c_l + i, r_t + j] \mid i \leq w, j \leq h]])} \ (f6)$$

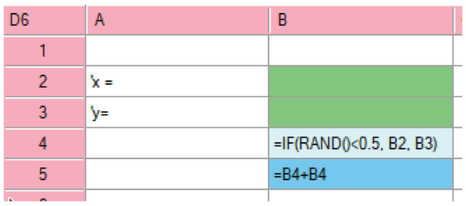
$$\frac{\begin{array}{c} \rho, \sigma \vdash e_1 \Downarrow v_1 \qquad \ldots \qquad \rho, \sigma \vdash e_n \Downarrow v_n \\ def(sdf) = (out, [in_1, \ldots, in_n], cells) \\ \rho' \text{ fresh} \qquad \rho'(in_1) = v_1 \qquad \ldots \qquad \rho'(in_n) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_n\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca) \end{array}}{\rho, \sigma \vdash sdf(e_1, \ldots, e_n) \Downarrow \rho'(out)} \ (f8)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow u_1 \qquad \ldots \qquad \rho, \sigma \vdash e_k \Downarrow u_k}{\rho, \sigma \vdash \texttt{CLOSURE}(sdf, e_1, \ldots, e_k) \Downarrow FunVal(sdf, [u_1, \ldots, u_k])} \ (f9)$$

$$\frac{\begin{array}{c} \rho, \sigma \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]) \\ \rho, \sigma \vdash e_1 \Downarrow v_1 \qquad \ldots \qquad \rho, \sigma \vdash e_n \Downarrow v_n \end{array}}{\rho, \sigma \vdash \texttt{CLOSURE}(e_0, e_1, \ldots, e_n) \Downarrow FunVal(sdf, [u_1, \ldots, u_k, v_1, \ldots, v_n])} \ (f10)$$

$$\frac{\begin{array}{c} \rho, \sigma \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]) \\ \rho, \sigma \vdash e_1 \Downarrow v_1 \qquad \ldots \qquad \rho, \sigma \vdash e_n \Downarrow v_n \\ def(sdf) = (out, [in_1, \ldots, in_{k+n}], cells) \\ \rho' \text{ fresh} \qquad \rho'(in_1) = u_1 \ \ldots \ \rho'(in_k) = u_k \\ \rho'(in_{k+1}) = v_1 \ \ldots \ \rho'(in_{k+n}) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca) \end{array}}{\rho, \sigma \vdash \texttt{APPLY}(e_0, e_1, \ldots, e_n) \Downarrow \rho'(out)} \ (f11)$$

Fig. 12. (*continued*)



D6	A	B
1		
2	'x =	
3	'y=	
4		=IF(RAND()<0.5, B2, B3)
5		=B4+B4

Fig. 13. Sheet-defined function `F` with input cells B2 and B3, output cell B5, and an intermediate cell B4.

and 8), display an error message, and indicate one or more cells involved in the cyclic dependency. This is somewhat weaker than our semantics, which would require the implementation to mark all involved cells (and those transitively dependent on them) with an error value such as `#CYCLE!`. Arguably, the behaviour of the major spreadsheet implementations is more than adequate, since knowing one cell on a cycle allows the user to debug and remove the cyclic dependency.

4. Implementation of the semantics

In this section, we briefly discuss a few aspects of how two implementations of Funcalc ensure that code adheres to the semantics.

Cells in Funcalc are evaluated by an interpreter and so follow the semantics closely.

Perhaps more interestingly, sheet-defined functions, in the standard implementation of Funcalc, are automatically compiled to Common Intermediate Language (CIL) bytecode. The intermediate and output cells in any sheet-defined function must produce the same value as in Fig. 13, where specifically the two references to cell B4 in the output cell B5 must both produce the same value. This is in accordance with rules (*e*8) and (*f*8) for sheet-defined function application and rules (*e*11) and (*f*11) for closure calls, whose bottom-most premise of each rule states that for each intermediate cell and output cell *ca*, we must find a value ρ′(*ca*) for the expression ϕ(*ca*) in that cell.

To abide by this semantic criterion, the compiler analyses the dependency graph of the sheet-defined function and records the number of references to its cells. If there is more than one reference, it emits code to store the result of the cell in a local variable, thus returning the same value each time it is used. Otherwise, if the cell is referenced only once, its formula is inlined at its unique occurrence in the generated bytecode.

In an extended version of Funcalc, a cost evaluator has been implemented [9]. The cost evaluator extends the interpreter for cell evaluation to sheet-defined functions.

The interpreter directly interprets the cells of a sheet-defined function by evaluating the output cell of a sheet-defined function and following dependencies back to its input cells. This requires proper abstraction of ρ: *Addr* → *Value*, the local cell environment or stack frame of a sheet-defined function as described in Section 3.4, in order to handle both recursive sheet-defined functions and normal function calls.

To implement ρ, we could directly modify the input cells of the sheet-defined function on each call. However, this would temporarily modify cells in the spreadsheet which could easily lead to inconsistencies. Instead, we keep track of an internal, local environment *lenv:*

Addr → *Value* that mimics ρ. When a sheet-defined function is called, we create and push a new local environment onto an internal stack and store the sheet-defined function's parameters in it by mapping the addresses of the input cells to their respective parameter values. This mimics the semantic rule for application (see (*e*8)) where the input parameters for the current function call are stored in ρ′ i.e. $\rho'(in_1) = v_1 ... \rho'(in_n) = v_n$. Upon invoking a recursive function call, we create and push a new local environment with the new parameters. When the recursive call returns, we pop the top-most local environment from the stack. Therefore, *lenv* behaves exactly like a stack frame following the intuition in Section 3.5.1. We still need one last detail for the local environment to work. Evaluation of a cell reference is modified to first look in the top-most local environment, if any, before examining the cells of the actual sheets. Thus when we do computation in recursive sheet-defined functions and need to evaluate an input parameter, we first look in the local environment and not in the actual spreadsheet.

We refer to [2] for the full details of the compiler and to our technical report [9] for more details on the relationship between the semantics and the implementation.

5. Conclusion and future work

In this paper we have presented a simple but precise operational semantics for the evaluation of extended spreadsheet formulas, with array formulas, sheet-defined functions and closures, as found in the Funcalc spreadsheet platform [2].

The evaluation semantics for simple Funcalc expressions was elaborated in Section 2 and semantics for extended spreadsheet expressions was developed in Section 3.

The semantics has served as a guideline for implementations and we have discussed two very different implementations; one based on runtime code generation and one based on interpretation. The latter forms the basis for an implementation of a cost evaluator, which can be used as a guide for load-balancing parallel computations in spreadsheets, e.g. via task partitioning for execution on multi-core CPUs [10] or offloading work to GPGPUs [11].

Given that major spreadsheet implementations (Microsoft Excel, LibreOffice Calc, Gnumeric, Google Sheets) do not explicitly describe their expected behaviour, it is impossible to prove that our semantics is the one intended by these implementations. It would, however, be an interesting future direction to experimentally verify if and how the semantic rules reflect such implementations and perhaps define alternative semantic rules for implementations that differ. Such rules could serve as inspiration for future updates from spreadsheet vendors or even serve as input for standardization efforts similar to the work on the C programming language [12]. Clearly for such work it would be necessary to extend the semantics to other types of values such as character strings and dates as discussed in Section 3.1.

Finally, one could imagine various tools, based in the formal semantics, for analyzing or verifying various aspects of spreadsheets. Verifying the correctness of a spreadsheet program may not seem such a big deal, but according to [13] it is generally accepted that nine out of every ten spreadsheets suffer some error. Bewig [13] reports on several cases where huge financial losses have been encountered due to incorrectness of spreadsheet programs. Thus based on the semantics presented in this paper, one could imagine a tool that could formally verify the correctness of the spreadsheet program.

Declaration of Competing Interests

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Peter Sestoft: Independent consultant at Microsoft Research Cambridge UK, Sep-Dec 2001.

Alexander Asp Bock: Previous employment as a research intern at Microsoft Research Cambridge (MSRC) for three months in 2017. The work conducted there is protected by a non-disclosure agreement.

The other authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Alexander Asp Bock: Conceptualization, Software, Writing - original draft, Writing - review & editing, Visualization. **Thomas Bøgholm:** Conceptualization, Software, Writing - review & editing. **Peter Sestoft:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Supervision, Validation, Writing - original draft. **Bent Thomsen:** Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Supervision, Writing - review & editing. **Lone Leth Thomsen:** Formal analysis, Investigation, Methodology, Supervision, Writing - review & editing.

References

[1] C. Scaffidi, Counts and earnings of end-user developers, 2017, https://www.linkedin.com/pulse/counts-earnings-end-user-developers-chris-scaffidi?published=t, (accessed 19 October 2017).
[2] P. Sestoft, Spreadsheet Implementation Technology, The MIT Press, 2014.
[3] S. Peyton-Jones, A. Blackwell, M. Burnett, A user-centred approach to functions in excel, Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Program, (2003), pp. 165–176. http://doi.acm.org/10.1145/944705.944721
[4] F. Nuñez, An extended spreadsheet paradigm for data visualisation systems, and its implementation, University of Cape Town, 2000 Ph.D. thesis.
[5] D. Wakeling, Spreadsheet functional programming, J. of Functional Program. 17 (2007) 131–143. https://doi.org/10.1017/S0956796806006186
[6] M. McCutchen, J. Borghouts, A. Gordon, S. Peyton-Jones, A. Sarkar, Elastic sheet-defined functions: generalising spreadsheet functions to variable-size input arrays, In submission (2019).
[7] G. Kahn, Natural Semantics, in: F.J. Brandenburg, G. Vidal-Naquet, M. Wirsing (Eds.), Proceedings of the Annul Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, 1987. 22–39
[8] H.R. Nielson, F. Nielson, Semantics with applications, An Appetizer, Springer-Verlag, 2007.
[9] A.A. Bock, T. Bøgholm, P. Sestoft, B. Thomsen, L.L. Thomsen, Concrete and abstract cost semantics for spreadsheets, IT University of Copenhagen and AAlborg University, Denmark, 2018 Technical report tr-2018-203.
[10] T. Bøgholm, K.G. Larsen, M. Muniz, B. Thomsen, L.L. Thomsen, Analyzing spreadsheets for parallel execution via model checking, Lect. Notes Comput. Sci. 11200 (2018).
[11] J. Trudeau, Collaboration and open source at AMD: Libreoffice, https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/, 2015 (accessed 31 July 2015).
[12] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R.N.M. Watson, P. Sewell, Into the depths of c: elaborating the de facto standards, ACM SIGPLAN Notices 51 (2016) 1–15.
[13] P.L. Bewig, How do you know your spreadsheet is right?, 2013, arXiv:1301.5878.

Alexander Asp Bock started his academic career as a Ph.D. student for Peter Sestoft as part of the Popular Parallel Programming (P3) project at the IT University of Copenhagen. His primary focus was on developing parallel evaluation strategies for accelerating spreadsheet computation for end-users. The work was implemented in the Funcalc research spreadsheet application. Later, he was a Postdoc working on further optimisations for Funcalc.

Thomas Bøgholm is an associate professor at the department of computer science at Aalborg University. His research interests include programming languages and program analysis in the areas of parallel programming and real-time embedded and safety critical systems, and works with object oriented program analysis and computational thinking.

Thomas has an M.Sc., cum laude, in Software Engineering, and a Ph.D. in computer science, from Aalborg University, Denmark.

Peter Sestoft is professor and head of the Computer Science Department at the IT University of Copenhagen. He works mainly with programming language technology, including functional, object-oriented and parallel programming languages, their use, implementation and optimization. He is a co-author, with Jones and Gomard, of the standard reference on partial evaluation (1993), and author of five other books, most recently Programming Language Concepts 2nd edition, Springer 2017.

Bent Thomsen is a Professor (MSO) in the department of computer science at Aalborg University, Denmark. His research is in programming languages and programming technology with applications in parallel, concurrent, distributed, mobile, embedded and data processing systems. He was Principal Researcher at ICL, UK and team-leader for the Facile Programming Language team at ECRC in Munich, Germany. Bent is a Chartered Engineer and a European Engineer (Eur Ing). He is a fellow of the British Computer Society. Bent has an M.Sc. in computer science from Aalborg University, Denmark, a Ph.D. and DIC from Imperial College, London University, UK.

Lone Leth Thomsen is an Associate Professor and study board chairman in the department of computer science at Aalborg University, Denmark. Her research interests include functional concurrent programming, web and IoT, parallelization of spreadsheets and future programming technology. She was Principal Researcher at ICL, UK and team-member in the Facile Programming Language team at ECRC, Germany. Lone is a Chartered Engineer and European Engineer (Eur Ing). She is a fellow of the British Computer Society. Lone has an M.Sc. in Software Engineering and Computing Systems from Aalborg University, Denmark, a Ph.D. and DIC from Imperial College, London University, UK.