Journal of Computer Languages 69 (2022) 101103

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

# On the cost semantics for spreadsheets with sheet-defined functions

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

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

ARTICLE INFO

*Keywords:*
Spreadsheets
Cost semantics
Abstract semantics
Funcalc
Sheet-defined functions
Recalculation

ABSTRACT

We give a simple operational cost semantics for evaluation of spreadsheet formulas and for full and minimal recalculation. We also present a semantics which caters for computing with unknown data values. This may be used to give an approximation of the cost when input data is not yet provided. This semantics is a rudimentary big-step trace-based abstract interpretation based upon the cost semantics. Together, the semantic presentations form the formal foundations for various cost calculations implemented in the Funcalc spreadsheet platform. This can be used in cost estimation tools, e.g. to estimate which formulas in a spreadsheet are the most expensive, or to schedule parallel recalculation of a spreadsheet. In future work, further analyzes and verification tools can be built upon the formal semantics to reduce the large number of errors that commonly occur in spreadsheets.

## 1. Introduction

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 it is almost impossible to predict or analyze the computational cost of spreadsheet computations.

Some complex spreadsheets may take a long time to recalculate. For instance, the `building-design` and `ground-water` benchmarks developed in connection with a study of parallelization of the LibreOffice spreadsheet program on AMD GPUs [2], have about one million data cells and 108,332 and 126,404 formula cells respectively and updates may take a long time. Other spreadsheets, such as `energy-market` with 534,507 formula cells, may take even longer.

As noted already by Mani Chandy in his 1985 keynote *Concurrent programming for the masses*, spreadsheets have a programming model that should be much easier to parallelize than traditional programming languages [3]. However, only sporadic efforts have been made in this area in the last 30 years [2,4–6].

The Popular Parallel Programming (P3) project[4] set out to investigate various approaches to parallelizing the execution of spreadsheet programs based on the open source spreadsheets Corecalc and Funcalc[5] implemented in C# and thoroughly described in [7]. The core idea is to view a spreadsheet as a program in a dataflow language, and then leverage and extend techniques for compilation of dataflow languages to shared-memory multicore machines, especially those techniques found in [8].

To realize the idea of parallel programming via spreadsheets, one approach is to adapt and further develop program analysis techniques to identify the parts of a spreadsheet that can be recalculated in parallel and subsequently find schedules for their execution, for example using model checking [9]. To find such schedules, it is necessary to estimate or calculate the computational cost of expressions in a spreadsheet, and in this paper we present an extension of the evaluation semantics presented in [7,10] to account for cost.

Sestoft's Funcalc spreadsheet platform [7] implements the notion of sheet-defined functions, inspired by Peyton-Jones et al. [11]. An example is given in Fig. 9 . A sheet-defined function is a user-defined function that can be defined directly in the spreadsheet cells of special function sheets using the same familiar formula syntax already known by end-users, without the need for external languages. Thus sheet-defined

* Corresponding author.
*E-mail addresses:* albo.researcher@gmail.com (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 IT-vest (grant number AAU-2018-59).
[3] Supported by Innovation Fund Denmark (grant number 7076-00029B), Projection of Balances and Benefits in Life Insurance (ProBaBLI) 2018–2022.
[4] https://www.itu.dk/people/sestoft/p3/.
[5] https://www.itu.dk/people/sestoft/funcalc/.

https://doi.org/10.1016/j.cola.2022.101103
Received 16 March 2021; Received in revised form 30 December 2021; Accepted 7 February 2022
Available online 23 February 2022

functions bring a natural abstraction mechanism to the world of spreadsheets. Funcalc also supports the notion of array formulas, as found in popular spreadsheet implementations such as Excel and OpenOffice Calc. Sheet-defined functions can be higher-order functions. Together with first class array formulas and a few simple built-in functions, such as map, reduce and fold, the expression language of Funcalc is an expressive, yet pure, higher-order functional programming language.

Inspired by Gomez et al. [12] and Rosendahl [13] we give a simple cost semantics for evaluation of a spreadsheet formula and for full and minimal recalculation of a spreadsheet. The cost semantics is a straightforward extension of our big-step semantics for spreadsheets [10]. However, the cost semantics differ in subtle ways from the original big-step semantics. We discuss the subtleties, especially their implications for implementations.

Analyzing or calculating cost or complexity of (higher-order) functional programming languages goes back to [14–17]. More recent work based on language semantics extended with some notion of cost have been presented in [18–23]. Both [24,25] introduce big-step cost semantics for functional programming languages and relate the lazy, respectively, the eager, evaluation strategies to implementations. A big-step cost semantics for a subset of the ML programming language is presented in [26], very closely akin to the cost semantics presented in this paper. The main difference between [24–26] and the work presented in this paper, apart from the application domain of spreadsheets, is the introduction of non-determinism in our cost semantics to allow for different implementations, e.g. allowing for evaluating arguments to functions in a left to right order or in parallel.

Both Gomez et al. and Rosendahl worked on cost translations for higher-order functional languages [12,13] which cater for computing with unknown data values. Adding such unknown values allows for a rudimentary abstract interpretation of programs which in many cases can provide a rather precise approximation of the actual cost of the computation. To provide a semantic foundation for calculation with unknown data values, we follow the ideas presented by Schmidt [27] and provide a big-step trace-based abstract interpretation for the cost semantics.

The cost semantics form the formal foundations for various cost calculations implemented in the Funcalc spreadsheet platform.

The rest of this paper is organized as follows: Section 2 presents the cost semantics, with 2.1 giving a cost semantics for simple spreadsheet formulas extending the semantics described in [7]. In Section 3 we give a cost semantics for Funcalc extended spreadsheet expressions and in Section 4 we give cost semantics for intrinsic functions. The extended evaluation semantics for Funcalc is further augmented to compute with unknown values in Section 5; this is a first step towards an approximate cost analysis based on abstract interpretation. Implementations for the concrete and abstract cost semantics are presented in Section 6. In Section 7, we present results for an initial implementation of a cost evaluator built upon the rules of our cost semantics. Conclusions and future work are presented in Section 8.

## 2. Cost semantics

In this section, we present a cost semantics for spreadsheet expressions which in addition to a computed value of the expression describes the possible cost of computing it. More precisely, the semantics describes the *work*, i.e. uni-processor cost [8], of the computation. In a parallel implementation, some of that work may be performed in parallel.

First, we give an operational cost semantics for simple spreadsheet expressions based on the evaluation semantics presented in [7] and extended in [10]. Then Section 3 extends the cost semantics to cover array formulas and sheet-defined functions.

In all cases the amount of work is described by a non-negative integer in $Nat_0$ representing some notion of computation step, for instance the number of rule applications, plus some measure of the cost of calling a built-in function (such as `SUM` over a range of cells). This notion of work can reasonably be assumed to be within a constant factor of the actual number of nanoseconds required to evaluate an expression.

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

**Fig. 1.** Syntax of the simplified formula language.

$$
\begin{array}{rcll}
n & \in & Number & = \{ \text{ IEEE floating} - \text{point numbers } \} \\
 & & Error & = \{ \texttt{\#DIV/0!}, \texttt{\#CYCLE!}, \texttt{\#NA} \} \\
ca & \in & Addr & = \{ \text{ cell addresses } (c,r) \} \\
v & \in & Value & = Number + Error \\
e & \in & Expr & = \{ \text{ formulas, see Figure 1 } \} \\
\phi & & & \in Addr \to Expr \\
\sigma & & & \in Addr \to Value
\end{array}
$$

**Fig. 2.** Sets and maps used in the spreadsheet semantics: $Number$ is the set of 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.

### 2.1. Cost semantics for simple formulas

For clarity of presentation, we start by giving a cost semantics for simple spreadsheet formulas as described in [7]. The simplified formulas used in this section are described in Fig. 1. 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 3.1.

Volatile functions, such as `RAND` in Fig. 1, are special functions that are unconditionally evaluated when recalculating. For instance, the function `RAND` produces random numbers and must be volatile to ensure that every recalculation produces a new random number even if the cell containing the volatile function is otherwise unchanged.

To describe the evaluation of formulas, we use the semantic sets and functions defined in Fig. 2. 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 but otherwise provide no additional semantic insight.

We use a map $\phi : Addr \to Expr$ 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 $\phi$ is the set of cell addresses that have a formula i.e. the set of non-blank cells $dom(\phi) = \{ ca \mid \phi(ca) \text{ is defined} \}$. The $\phi$ function is not affected by recalculation, only by editing the sheet.

The result of a recalculation is modeled by function $\sigma : Addr \to Value$, where $\sigma(ca)$ is the computed value in cell $ca$. The $\sigma$ function gets updated by each recalculation (see Section 2.3).

It is quite straightforward to extend the evaluation semantics rules in [7,10] to the new cost semantics rules given in Fig. 3.

The evaluation judgment $\sigma \vdash e \Downarrow v$ gets extended to $\sigma \vdash e \Downarrow v, c$ where $v$ is a computed value of the expression $e$ and $c$ is the cost of

$$\frac{ca \notin dom(\sigma)}{\sigma \vdash \texttt{ca} \Downarrow 0.0, 1} \; (c2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma \vdash \texttt{ca} \Downarrow v, 1} \; (c2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \in Error}{\sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v_1, 1 + c_1} \; (c3e)$$

$$\frac{\sigma \vdash e_1 \Downarrow 0.0, c_1 \qquad \sigma \vdash e_3 \Downarrow v, c_3}{\sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_3} \; (c3f)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \notin \{0.0\} \cup Error \qquad \sigma \vdash e_2 \Downarrow v, c_2}{\sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_2} \; (c3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma \vdash \texttt{RAND}() \Downarrow v, 1} \; (c4)$$

$$\frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\ \sigma \vdash e_j \Downarrow v_j, c_j \qquad v_i \in Error \text{ for some } i \in J \end{array}}{\sigma \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow v_i, 1 + \sum_{j \in J} c_j} \; (c5e)$$

$$\frac{\begin{array}{c} \sigma \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma \vdash e_n \Downarrow v_n, c_n \\ \forall i.\ v_i \notin Error \end{array}}{\sigma \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n), 1 + \sum_{j=1,n} c_j + work(f, v_1, \ldots, v_n)} \; (c5v)$$

**Fig. 3.** Cost (or work) semantics rules for simplified spreadsheet formulas.

computing that value. This judgment states that when $\sigma$ describes the calculated values of all cells, then formula $e$ may evaluate to value $v$ at computational cost $c$. As in [10], the semantics is nondeterministic ("may") in the sense that the evaluation of an expression $e$ could produce many different values $v$ at many different costs $c$. The cost of evaluating $e$ is given under the assumption that all referred-to cells are already computed.

The formula evaluation rules in Fig. 3 are explained as follows:

Rule (c1) says that evaluating a number constant `n` requires 1 computation step, and similarly for cell references by rules (c2b) and (c2v).

Rule (c3e) says that if $e_1$ may evaluate to error $v_1$ in $c_1$ computation steps, then `IF`$(e_1,e_2,e_3)$ may evaluate to error $v_1$ in $1 + c_1$ computation steps.

Rule (c3f) says that if $e_1$ may evaluate to the non-error number 0.0 in $c_1$ computation steps and the "false branch" $e_3$ may evaluate to $v$ in $c_3$ computation steps, then `IF`$(e_1,e_2,e_3)$ may evaluate to value $v$ in $1 + c_1 + c_3$ computation steps.

Rule (c3t) is similar to rule (c3f) for when $e_1$ may evaluate to some non-error non-zero number $v_1$ in $c_1$ computation steps. 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; and also this rule reflects spreadsheet reality.

Rule (c4) says that function call `RAND()` may evaluate to any (non-error) number $v$ greater than or equal to zero and less than one, in one computation step.

Rule (c5e) says that an implementation may choose to evaluate just a subset $\{e_j \mid j \in J\}$ of the arguments when some $e_i$ with $i \in J$ evaluates to an error $v_i$, and then let $v_i$ be the result of the function call. Also, it says that the total cost of this is the cost $\sum_{j \in J} c_j$ of evaluating that subset of arguments, plus one. The rationale for this is discussed in Section 2.2.

Rule (c5v) says that if each argument $e_i$ may evaluate to a non-error value $v_i$ in $c_i$ computation steps and applying the actual function $f$ to argument values $(v_1, \ldots, v_n)$ produces value $v$ at a cost of $work(f, v_1, \ldots, v_n)$ computation steps, then the call `F`$(e_1, \ldots, e_n)$ may evaluate to value $v$ using a total of $1 + \sum_{j=1,n} c_j + work(f, v_1, \ldots, v_n)$ computation steps. Here $work(f, v_1, \ldots, v_n)$ describes the cost of applying function $f$ to argument values $(v_1, \ldots, v_n)$. For instance, one would expect $work(+, v_1, v_2) = 1$ since the cost of addition is independent of the actual numbers added. By contrast, for functions on array values one would expect the cost to depend on the argument array size; for instance, $work(transpose, v_1) = w \cdot h + 1$ when array value $v_1$ has $w$ columns and $h$ rows. This will be defined in Section 4 when we discuss Funcalc's intrinsic functions.

Making each cost rule add 1 to the cost incurred by subexpression evaluations may appear very simplistic. It means that the cost semantics essentially counts the number of rule applications and ensures that costs increase monotonically. A more realistic cost semantics might replace each occurrence of "1" with a suitable constant indicating a number of nanoseconds for the operation, such as 1 for evaluating a constant, 8 for evaluating a cell reference, 40 for a call to `RAND`, and similar. However, if we are interested in cost up to a constant factor, counting the number of rule applications works just as well, and avoids some

notational clutter. Also, the real time cost of something as simple as a cell reference may vary from 1 ns to 80 ns depending on whether the relevant data is already in the CPU hardware cache or not.

### 2.2. Rationale for cost of an error argument

While most of the cost semantics rules in Fig. 3 are obvious extensions of the evaluation rules presented in [10], this is not the case for rule (c5e) for unsuccessful function call evaluation. Here we explain why.

It is possible to imagine a cost rule (c5bad) like this:

$$\frac{\sigma \vdash e_i \Downarrow v_i, c_i \qquad v_i \in Error}{\sigma \vdash \mathrm{F}(e_1, \ldots, e_n) \Downarrow v_i, 1 + c_i} \; (c5bad)$$

This rule says that if some argument $e_i$ may evaluate to an error $v_i$ using $c_i$ computation steps, then the call $\mathrm{F}(e_1, \ldots, e_n)$ to a function F may evaluate to error $v_i$ in $1 + c_i$ computation steps. However, this cost is unrealistically low: a conforming implementation would have to correctly guess which (if any) argument expression $e_i$ can evaluate to an error, and then evaluate only that expression. Such an implementation would seem implausibly clever.

A more realistic rule might stipulate instead that the cost is (at least) the sum of the costs of evaluating all argument expressions. This corresponds to implementations that would evaluate all arguments before checking whether any of them evaluates to an error. However, this is needlessly pessimistic since an implementation may stop evaluating arguments once one of them evaluates to an error.

Another realistic cost rule might correspond to implementations that evaluate argument expressions $e_1, e_2, \ldots$ from left to right until one of them (if any) evaluates to an error. However, this restricts the possible implementations and would preclude or complicate parallel evaluation of arguments.

Instead we propose rule (c5e) in Fig. 3 which corresponds to implementations that may evaluate the argument expressions in any order (or in parallel) but may avoid evaluating all of them in case one evaluates to an error. As shown in the rule this corresponds to choosing a subset $J \subseteq \{1, \ldots, n\}$ of the argument indices and evaluating only the $e_j$ for which $j \in J$, to values $v_j$ at costs $c_j$, where one of the $v_j$ is an error, and then stating that the total cost of the call is the sum $\sum_{j \in J} c_j$ of the costs of the arguments actually evaluated, plus one.

Since the set $J$ may be chosen in many ways, this rule introduces nondeterminism in the evaluation cost, in addition to nondeterminism in the computed value. Note also that rule (c5e) encompasses all three alternative rules discussed above, by choosing $J = \{i\}$ as the singleton set for which $v_i$ is an error (using unrealistically perfect foresight), or $J = \{1, \ldots, n\}$ to evaluate all arguments, or $J = \{1, \ldots, i\}$ as the least prefix of argument indexes for which $v_i$ is an error.

### 2.3. Cost of simple recalculation and consistency

Sections 2.1 and 2.2 above gave evaluation-and-cost rules for evaluation of individual spreadsheet formulas. How do we describe the cost of a full recalculation or minimal recalculation in terms of these? First, we introduce a cost environment $\gamma : Addr \to Nat_0$ such that $\gamma(ca)$ is the cost of evaluating the formula at cell address $ca$. Using this cost environment, we can now express the cost of a *full recalculation* of a spreadsheet described by $\phi$ and $\sigma$ as the cost of evaluating the formula of every non-blank cell once:

$$fullcost = \sum_{ca \in dom(\phi)} \gamma(ca)$$

The purpose of a full recalculation is to compute a consistent spreadsheet: one in which the computed value of each non-blank cell agrees with its formula. This is formalized in Fig. 4.

Requirement (1) says that a recalculation must find a value $\sigma(ca)$, possibly an error, as well as a cost $\gamma(ca)$, for every non-blank cell $ca$.

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

**Fig. 4.** Recalculation consistency requirements for simple formulas with cost.

$$\begin{array}{ll} (1) & dom(\sigma') = dom(\gamma') = dom(\phi) \\ (2) & \forall ca \in dom(\phi).\ \sigma' \vdash \phi(ca) \Downarrow \sigma'(ca), \gamma'(ca) \\ (3) & \forall ca \notin dirty(ca_0).\ \sigma'(ca) = \sigma(ca) \end{array}$$

**Fig. 5.** Minimal recalculation consistency requirements for simple formulas with cost.

Requirement (2) says that the computed value $\sigma(ca)$ and cost $\gamma(ca)$ must agree with the cell's formula $\phi(ca)$ for every non-blank cell $ca$, thus asserting that the spreadsheet's cell values as described by $\sigma$ are consistent with each other.

To express the cost of a *minimal recalculation* initiated by editing a single cell $ca_0$ in a consistent spreadsheet represented by $\phi$ and $\sigma$, we need the set $dirty(ca_0)$ of cells that must be recalculated after cell $ca_0$ has been edited. This set is defined in terms of the "supports" or "precedent" relation, where cell $ca_a$ is said to support (or be a precedent of) cell $ca_b$ if $ca_b$ directly depends on $ca_a$, by $ca_b$'s formula containing a reference to $ca_a$.

Now $dirty(ca_0)$ is simply the transitive closure, under the "supports" relation, of the set containing cell $ca_0$ and every cell whose formula is volatile. Since $dirty(ca_0)$ is defined via the "supports" relation it may be an overapproximation of the set of cells that really need to be evaluated in a minimal recalculation. Namely, if cell $ca_a$ supports cell $ca_b$ but a recalculation of $ca_a$ happens not to change its value, cell $ca_b$ might not really need to be recalculated, but $dirty(ca_a)$ would nevertheless contain $ca_b$.

The consistency requirements following a minimal recalculation, shown in Fig. 5, refine those for full recalculation. There must be consistent spreadsheet values $\sigma$ before the minimal recalculation; we again impose requirements (1) and (2) on the spreadsheet values $\sigma'$ and costs $\gamma'$ after the minimal recalculation. In addition, we require (3) that all cells not recalculated retain their values.

With these definitions, the total cost of a minimal recalculation after a change to cell $ca_0$ is the sum of the costs of evaluating the cells in $dirty(ca_0)$:

$$minimalcost = \sum_{ca \in dirty(ca_0)} \gamma'(ca)$$

Note also that the consistency requirements and cost for full and minimal recalculation intentionally do not specify how recalculation actually proceeds, but specify only the requirements that must hold for each cell after recalculation.

## 3. Cost semantics for extended formulas

In this section we extend the cost semantics to cover array formulas and sheet-defined functions.

### 3.1. Extended expressions and semantic sets

The simple spreadsheet cost semantics from Section 2.1 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 modeled as an underlying formula $ae$ which is itself just an expression, expected to evaluate to an array

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

**Fig. 6.** Syntax of the Funcalc extended formula language, with six 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 $sdf$, further application of a closure $e_0$, and full application of a closure $e_0$.

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

**Fig. 7.** Sets and maps used in the Funcalc extended spreadsheet semantics. There are the following differences relative to Fig. 2: $av \in ArrVal$ is an array value with $w \cdot h$ component values $v_{ij}$, where indices $i,j$ are one-based column indices and row indices respectively in keeping with "A1-style" cell references, where the column A is given first, the row second. Indices must be positive; $fv \in FunVal$ is a function value (closure) consisting of a function name $sdf$ and $0 \le k \le arity(sdf)$ given argument values $u_i$. Now, $v \in Value$ is either a number, an error, an array value or a 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: $\alpha$ maps an array expression $ae$ to its value, and $\rho$ maps a function sheet cell address to its value.

of values, called an array value in Funcalc terminology. That array value's components are distributed over a target cell area, with one such component in each cell.

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 $0 \le k \le arity(sdf)$; see Fig. 7. Funcalc supports early-bound arguments in any argument position, using the NA function, which returns the #NA "not available" error, as a placeholder for late-bound arguments, as in `=LOG(NA(), 2)`. One could model this behavior by also recording the argument position along with the value, but for simplicity we use a prefix here, as modeling the full behavior leads to a more complicated semantics without significant additional insight.

A closure is created by calling the CLOSURE built-in with a sheet-defined function $sdf$ and 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 $k + n = arity(sdf)$, and will execute the underlying sheet-defined function.

The cost semantics for Funcalc extended spreadsheet formulas is given by judgments of the form $\sigma,\alpha \vdash e \Downarrow v,c$ which say that when $\sigma$ describes cell values and $\alpha$ describes array expression values, expression $e$ may evaluate to value $v$ at a cost of $c$ computation steps. The rules defining these judgments are given in Fig. 8.

The extended cost semantics rules in Fig. 8 draw on the simple cost semantics rules in Fig. 3.

Rules (g1) through (g5v) are very similar to the simple cost semantics rules (c1) through (c5v). This includes the somewhat complicated case (g5e) of a function argument evaluating to an error, explained in Section 2.2.

Rule (g6) states that the cost of evaluating a sheet cell area expression that produces an array value of $w$ columns and $h$ rows is $w \cdot h$, the number of components in the resulting array value, plus one. The notation $\sigma[c_l + i, r_t + j]$ denotes indexing into the sheet $\sigma$ using $i$ and $j$ as offsets from the top left cell $(c_l, r_t)$ of the cell area.

Rule (g7) states that the cost of evaluating a cell that is part of an array formula is 1. This is because we require the array formula's shared underlying array expression to be evaluated at most once in a recalculation, so that evaluating the cell is just a matter of indexing into the resulting array.

Rule (g8) states that the cost of calling a sheet-defined function is the cost of evaluating all arguments, plus the cost of evaluating the function body, plus one. We use a helper function *def* that returns the addresses of the output cell *out*, input cells $[in_1,\ldots,in_n]$, and intermediate cells *cells* that compute intermediate values in the function. Each call, also each recursive call, has its own fresh $\rho'$ and $\gamma'$ environments that are both ephemeral: there is no way to refer to a function sheet cell value after the function has returned. Hence these environments are similar to a stack frame in ordinary programming language implementation. The cost of evaluating the function body is the sum of the costs of evaluating the cells used to define the function, as described by the cost environment $\gamma'$. Here, $dom(\gamma')$ must equal $dom(\rho') \setminus \{in_1,\ldots,in_n\}$ which includes the output cell, but there is some

$$\frac{}{\sigma, \alpha \vdash \texttt{n} \Downarrow n, 1} \; (g1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma, \alpha \vdash \texttt{ca} \Downarrow 0.0, 1} \; (g2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma, \alpha \vdash \texttt{ca} \Downarrow v, 1} \; (g2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \in Error}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v_1, 1 + c_1} \; (g3e)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow 0.0, c_1 \qquad \sigma, \alpha \vdash e_3 \Downarrow v, c_3}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_3} \; (g3f)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \neq 0.0 \qquad \sigma, \alpha \vdash e_2 \Downarrow v, c_2}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_2} \; (g3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma, \alpha \vdash \texttt{RAND}() \Downarrow v, 1} \; (g4)$$

$$\frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\ \sigma, \alpha \vdash e_j \Downarrow v_j, c_j \qquad v_i \in Error \text{ for some } i \in J \end{array}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow v_i, 1 + \sum_{j \in J} c_j} \; (g5e)$$

$$\frac{\begin{array}{c} \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ \forall i.\ v_i \notin Error \end{array}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n), 1 + \sum_{j=1,n} c_j + work(f, v_1, \ldots, v_n)} \; (g5v)$$

$$\frac{\begin{array}{c} (c_1, r_1) = ca_1 \qquad (c_2, r_2) = ca_2 \qquad (c_l, c_r) = sort(c_1, c_2) \qquad (r_t, r_b) = sort(r_1, r_2) \\ w = c_r - c_l + 1 \qquad h = r_b - r_t + 1 \end{array}}{\sigma, \alpha \vdash ca_1 : ca_2 \Downarrow ArrVal(w, h, [[\sigma[c_l + i, r_t + j] \mid 0 \leq i < w, 0 \leq j < h]]), 1 + w \cdot h} \; (g6)$$

**Fig. 8.** Cost (or work) semantics rules for Funcalc extended spreadsheet formulas. The corresponding consistency requirements on recalculation are given in Fig. 12.

flexibility in exactly which set of cells $dom(\rho')$ should be evaluated. See the discussion in Section 3.2.

Rule (g9) states that the cost of creating a closure is the cost of evaluating the $k$ given arguments, plus one.

Rule (g10) states that the cost of partially applying a closure is the cost of evaluating the first argument $e_0$ to a function value with $k$ early-bound arguments, plus the cost of evaluating the $n$ given arguments to the call to CLOSURE, plus one. This may evaluate to a new function value with $k + n$ arguments.

Rule (g11) states that the cost to call a closure is the cost of evaluating the closure expression, plus the cost of evaluating the remaining arguments, plus the cost of evaluating the called function's body, plus one. Similar to rule (g8), $dom(\gamma')$ must equal $dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}$, and Section 3.2 discusses how to choose $dom(\rho')$ and hence $dom(\gamma')$.

### 3.2. The cost of calling a sheet-defined function

The rules (g8) and (g11) for calling a sheet-defined function leave unspecified the set $dom(\rho')$ of the function's cells that should be evaluated, and hence the set $dom(\gamma') = dom(\rho') \setminus \{in_1, \ldots, in_n\}$ whose evaluation costs should be included in the call cost.

The set $dom(\rho')$ may contain all the function's cells, but it suffices to include only those cells actually needed to compute the value of the output cell $out$. For an ordinary semantics this distinction is less important, since it does not affect the result $\rho'(out)$ of the function call, assuming that evaluation terminates. However, for the cost semantics the distinction is crucial. Obviously, evaluating cells that are not needed, and hence including them in $dom(\rho')$ and in $dom(\gamma')$, affects the cost $\sum_{ca \in dom(\gamma')} \gamma'(ca)$ of the computation.

If the value of a cell $ca$ is needed, directly or indirectly, to compute the value of the output cell $out$, then $ca$ must be in $dom(\rho')$. Conversely, a cell whose value is not needed by the output cell should not be

$$\frac{}{\sigma, \alpha \vdash ae[i,j] \Downarrow \alpha(ae)[i,j], 1} \quad (g7)$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ def(sdf) = (out, [in_1, \ldots, in_n], cells) \\ \rho', \gamma' \text{ fresh} \quad \rho'(in_1) = v_1 \quad \ldots \quad \rho'(in_n) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_n\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca)\end{array}}{\sigma, \alpha \vdash sdf(e_1, \ldots, e_n) \Downarrow \rho'(out), 1 + \sum_{j=1,n} c_j + \sum_{ca \in dom(\gamma')} \gamma'(ca)} \quad (g8)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow u_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_k \Downarrow u_k, c_k}{\sigma, \alpha \vdash \texttt{CLOSURE}(sdf, e_1, \ldots, e_k) \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), 1 + \sum_{j=1,k} c_j} \quad (g9)$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_0 \\ \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n\end{array}}{\sigma, \alpha \vdash \texttt{CLOSURE}(e_0, e_1, \ldots, e_n) \Downarrow FunVal(sdf, [u_1, \ldots, u_k, v_1, \ldots, v_n]), 1 + c_0 + \sum_{j=1,n} c_j} \quad (g10)$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_0 \\ \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ def(sdf) = (out, [in_1, \ldots, in_{k+n}], cells) \\ \rho', \gamma' \text{ fresh} \quad \rho'(in_1) = u_1 \ \ldots \ \rho'(in_k) = u_k \\ \rho'(in_{k+1}) = v_1 \ \ldots \ \rho'(in_{k+n}) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca)\end{array}}{\sigma, \alpha \vdash \texttt{APPLY}(e_0, e_1, \ldots, e_n) \Downarrow \rho'(out), 1 + c_0 + \sum_{j=1,n} c_j + \sum_{ca \in dom(\gamma')} \gamma'(ca)} \quad (g11)$$

**Fig. 8.** (*continued*).

| A10 | A | B |
|---|---|---|
| 1 | =DEFINE("fct", B5, B2, B3) | |
| 2 | 'p = | |
| 3 | 'n = | |
| 4 | | =B2*B2*B3*B3 |
| 5 | 'res = | =IF(RAND()<B2, B3, B4) |

**Fig. 9.** A sheet-defined function. The `DEFINE` call in cell A1 says that the sheet-defined function's name is FCT, its output cell is B5 (colored dark blue) and its input cells are B2 and B3 (colored green). When the function is called elsewhere, as in `FCT(0.5, 7)`, the argument values 0.5 and 7 are copied to the input cells B2 and B3, and the output cell B5 is evaluated to obtain the function's result, which here is either 7 or 0.5·0.5·7·7 = 11.75, depending on the result of `RAND()` in B5. Note that the intermediate cell B4 (colored light blue) needs to be evaluated only if the condition in B5 is false. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in $dom(\rho')$. However, whether a cell $ca$ is needed or not cannot be determined prior to evaluation. It depends both on the input cell values (the function's argument values) and on the evaluation of volatile functions such as `RAND`. Consider the example sheet-defined function FCT in Fig. 9.

If input B2 ≥ 1, the condition in output cell B5 is always true and the function evaluates to B3 without having to evaluate B4; and if B2 ≤ 0, the condition in B5 is always false, and cell B4 must be evaluated to produce the result of the function.

When 0 < B2 < 1, the value of `RAND()` determines whether cell B4 really needs to be evaluated. A reasonable cost semantics should allow for leaving out the cost of evaluating cell B4 when its value is not needed. On the other hand, it should also allow for adding in that cost, so as to correctly describe an implementation that speculatively evaluates B4 although its value may not be needed. For instance, an implementation may evaluate B4 to exploit available parallel computation resources, or simply because the cost of unconditionally evaluating B4 is smaller than the cost of determining whether its value is needed (and then performing the relevant conditional jumps, synchronization, or the like).

Thus in the semantics there should be some freedom in choosing $dom(\rho')$ and hence $dom(\gamma')$ and hence the total cost of the function call. The choice should be subject to a consistency requirement: if cell $ca \in dom(\rho')$ and the value of $ca$ depends on cell $ca_r$, then $ca_r \in dom(\rho')$ too.

How can we describe more formally that the value of a cell $ca_r$ is needed to compute the output cell and hence the function's return value? In the Funcalc implementation, so-called evaluation conditions [7, Chapter 9] are used to control which cells must be evaluated. However, that is a particular *implementation mechanism* and should not be part of the cost semantics *specification*.

Hence we propose to specify the consistency requirement as follows. Consider an application of rule (g8) and all the inference trees that prove the judgments in the last row of premises. The domains $dom(\rho')$ and hence $dom(\gamma') = dom(\rho') \setminus \{in_1, \ldots, in_n\}$ must satisfy the following. For each $ca \in dom(\rho') \setminus \{in_1, \ldots, in_n\}$ there is an inference tree that proves

$$\rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca)$$

Now the consistency requirement says that for each function-sheet cell reference $ca_r \in cells$ encountered while building that inference tree, it is the case that $ca_r \in dom(\rho')$. In other words, any (non-input)

| A10 | A | B |
|---|---|---|
| 1 | =DEFINE("ex", B5, B2, B3) | |
| 2 | 'p = | |
| 3 | 'n = | |
| 4 | | =EX(B2, B3+1) |
| 5 | 'res = | =IF(RAND()<B2, B3, B4) |

**Fig. 10.** A sheet-defined function EX such that EX($p$,1) returns a random sample (1, 2, ...) from the geometric distribution with parameter $p$. The function definition is similar to FCT in Fig. 9, but cell B4 contains a recursive call to EX itself, so now it is essential that cell B4 does not get evaluated unconditionally. By eventually evaluating B4 only when it is needed, we can achieve that a call EX($p$, $n$) terminates if and only if $p > 0$.

$$\frac{ca \in dom(\rho) \qquad \rho(ca) = v}{\rho, \sigma \vdash \texttt{ca} \Downarrow v, 1}\ (h2f)$$

**Fig. 11.** Example cost semantics rule for Funcalc sheet-defined functions.

$$\begin{array}{ll} (1) & dom(\sigma) = dom(\phi) \\ (2) & \forall ca \in dom(\phi).\ \sigma, \alpha \vdash \phi(ca) \Downarrow \sigma(ca), \gamma(ca) \\ (3) & \forall ae \in dom(\alpha).\ \sigma, \alpha \vdash ae \Downarrow \alpha(ae), \gamma(ae) \\ (4) & dom(\gamma) = dom(\phi) \cup dom(\alpha) \end{array}$$

**Fig. 12.** The consistency requirements on recalculation and cost with array formulas and sheet-defined functions. The judgment $\sigma, \alpha \vdash e \Downarrow v, c$ is defined in Fig. 8. An array expression $ae$ is just an (array-valued) expression $e$, evaluated the same way.

cell $ca_r$ referred to during the evaluation of the sheet-defined function must have a value, meaning $ca_r \in dom(\rho')$, so that cell lookup succeeds for a cell reference $ca_r$ inside a sheet-defined function. Also, the cost of that computation must be accounted for, meaning $ca_r \in dom(\gamma')$.

Note that this consistency requirement is general enough to allow for speculative computation of unneeded cells, so long as this does not lead to an attempt to build an infinite inference tree, representing nonterminating recursion.

To illustrate the subtlety of the choice of whether to evaluate an unneeded cell, consider function EX in Fig. 10. This is a slight variant of FCT, where the decisive difference is that the trivial formula in B4 has been replaced with a recursive call `=EX(B2, B3+1)`, so that now it is essential both for termination and correct cost accounting that B4 is evaluated only when needed.

### 3.3. Cost semantics for function sheets

A cost semantics for sheet-defined functions on function sheets can be given by rules defining judgments of the form $\rho, \sigma \vdash e \Downarrow v, c$. Such judgments are referred to in rules (g8) and (g11). Because the rules are simple extensions of the cost semantics in Fig. 8, we give only one of these rules here, in Fig. 11.

### 3.4. Cost of extended recalculation

The cost of recalculation for Funcalc extended formulas must account for array formulas and for sheet-defined functions.

The cost of evaluating the array expression $ae$ underlying an array formula is defined as for any other expression. We use the $\gamma$ environment also to record this cost as $\gamma(ae)$, so its type is now $\gamma : Addr + Expr \to Nat_0$. The consistency requirements for a cost semantics accounting also for array formulas are shown in Fig. 12.

The total cost of a full recalculation therefore is the sum of computing the formula in every cell, plus the cost of computing the array expression underlying every array formula:

$$fullcost = \sum_{ca \in dom(\phi)} \gamma(ca) + \sum_{ae \in dom(\alpha)} \gamma(ae)$$

We extend the $dirty(ca_0)$ set to also include array expressions that need to be recalculated (in addition to non-array expression cells that also need to be recalculated), so now $dirty(ca_0) \subseteq Addr + Expr$. Hence the cost of a minimal recalculation can be expressed as before:

$$minimalcost = \sum_{ca \in dirty(ca_0)} \gamma'(ca)$$

where $\gamma'$ is a cost environment determined in a similar manner as in Section 2.3.

## 4. Rules for intrinsic functions

In this section, we extend the operational cost semantics from Section 3 by expanding the function application rule (g5v) for a meaningful subset of intrinsic functions in Funcalc. Recall rule (g5v), repeated below for convenience.

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \qquad \forall i. v_i \notin Error}{\sigma, \alpha \vdash \mathrm{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n), 1 + \sum_{j=1,n} c_j + work(f, v_1, \ldots, v_n)}\ (\text{g5v})$$

By "meaningful subset" we mean that it is not sensible or interesting to give rules for some of the intrinsic functions. For example, EXTERN returns the result of a call to an external library. While the returned value can be (and is) given by a plain C# `object` type, its cost is undefined. The call may perform any operation from querying a database to initiating some long-running, unknown computation that we have insufficient knowledge to approximate. Alternatively, we could give meaningful rules for some common uses for EXTERN such as the methods in the .NET libraries, but we forgo this here. In our presentation, we focus only on ordinary, interpreted sheets, as the rules for function sheets are mostly analogous. As a starting point, consider the rule for the SIN function that computes the sine of its input value.

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \in Number}{\sigma, \alpha \vdash \mathtt{SIN}(e) \Downarrow sin(v), 1 + c}\ (\text{sin})$$

The rule states that if the expression $e$ may evaluate to a number $v$ at cost $c$ then the function application expression SIN(e) may evaluate to the actual function application $sin(v)$ at total cost $1 + c$. Similar rules can be given for COS and TAN.

We introduce a few conventions for array values that must be borne in mind when reading the semantic rules in the following sections. First, we use a more compact notation for array values:

$$Av(w, h, [[v_{ij}]]) \triangleq ArrVal(w, h, [[v_{ij} \mid i \le w, j \le h]])$$

To refer to an entire column or row of an array value, we use notation such as $[[av_{i*}]]$ to refer to column $i$. Thus, $[[av_{1*}]]$ refers to the first column of an array value whereas $[[av_{*3}]]$ refers to the third row.

We define concatenation operators for array values $au$ and $av$. The horizontal concatenation $au : av$ is the array consisting of $au$'s columns on the left followed by $av$'s columns on the right; here $au$ and $av$ must have the same number of rows. The vertical concatenation $au; av$ is the array consisting of $au$'s rows on the top followed by $av$'s rows below; here $au$ and $av$ must have the same number of columns. These operators are associative and have suitable zero-column or zero-row arrays as units. The concatenation operators do not produce nested array values: concatenating two array values creates a single array value with the elements from both.

Additionally, we can construct an array value from a set of values that may produce nested array values: $[[v_1, v_2, v_3]]$. Each value of the

$$\frac{v \in Number}{\sigma, \alpha \vdash \texttt{NOW}() \Downarrow v, 1} \text{ (now)}$$

$$\frac{}{\sigma, \alpha \vdash \texttt{PI}() \Downarrow \pi, 1} \text{ (pi)}$$

$$\frac{}{\sigma, \alpha \vdash \texttt{NA}() \Downarrow \texttt{\#NA}, 1} \text{ (na)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \in Number}{\sigma, \alpha \vdash \texttt{ABS}(e) \Downarrow |v|, 1 + c} \text{ (abs)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \in Number}{\sigma, \alpha \vdash \texttt{ASIN}(e) \Downarrow asin(v), 1 + c} \text{ (asin)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v = 0}{\sigma, \alpha \vdash \texttt{NOT}(e) \Downarrow 1, 1 + c} \text{ (not-1)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \neq 0}{\sigma, \alpha \vdash \texttt{NOT}(e) \Downarrow 0, 1 + c} \text{ (not-2)}$$

$$\frac{\begin{array}{cc}\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 \\ v_1 \in Number & v_2 \in Number\end{array}}{\sigma, \alpha \vdash \texttt{CEILING}(e_1, e_2) \Downarrow ceiling(v_1, v_2), 1 + c_1 + c_2} \text{ (ceiling)}$$

**Fig. 13.** Operational cost semantics for a subset of Funcalc's built-in first-order functions. Error cases are omitted, as are trivially similar cases e.g. we show only `ASIN` and not also `COS`, `ACOS` and so on.

new array value is exactly as given so if $v_2$ is an array value itself, it will be nested inside of the constructed array value instead of its values being concatenated.

In general, we mostly omit the "error rules" dealing with cases where a function argument evaluates to an error value, and ask the reader to imagine analogs of rule (g5e) in Section 3. For most functions, we assume all arguments are non-error values and that we evaluate them in some order according to a desired implementation. For full details of all functions available in Funcalc, we refer the reader to [7]. First-order intrinsic functions are given in Section 4.1, higher-order functions are given in Section 4.2.

### 4.1. Rules for first-order intrinsic functions

The rules for Funcalc's first-order intrinsic functions are given in Fig. 13 in the context of ordinary sheets.

Rule (now) has one premise stating that if the result may evaluate to a number, the call may evaluate to value $v$ at cost 1 where $v$ is the number of fractional days since 30 December 1899.

Rule (pi) states that `PI()` may evaluate to the mathematical constant $\pi$ at cost 1.

Rule (na) states that the function application `NA()` may evaluate to the error value `#NA` at cost 1. This is used to indicate that a value is not available or to indicate a late-bound parameter in a partially applied closure.

Rule (abs) states that if $e$ may evaluate to the number $v$ at cost $c$ then the call may evaluate to the absolute value of $v$.

Rule (asin) is similar to rule (abs) but may instead evaluate to the result of a call to the actual inverse trigonometric function *asin*.

Rules (not-1) and (not-2) handle the two different outcomes of the `NOT` function. Rule (not-1) states that if $e$ may evaluate to zero (false) at some cost then the call may evaluate to one (true); rule (not-2) handles the opposite case.

Rule (ceiling) states that if the two argument expressions may evaluate to numbers then the expression may evaluate to $v_1$ rounded to $v_2$ decimal digits, as more precisely specified by the underlying *ceiling* function.

Rule (equal) is akin to rule (ceiling) except that it may evaluate to the equality comparison between the two numbers. We leave out the rules for other comparisons.

Rules (and-false) and (and-true) are inspired by rule (c5e) in Fig. 3, using reasoning analogous to that in Section 2.2. The point is that `AND` may use short-cut, speculative or parallel evaluation: once it finds that some argument $e_j$ evaluates to 0 (false), it may skip evaluating all the other arguments and immediately return 0. However, an implementation cannot reasonably be expected to guess beforehand which $e_j$ may evaluate to zero and evaluate only that, so the total cost should allow for the evaluation of some subset $J$ of arguments. Rule (and-false) says that if one can pick a subset $J$ of indices and there exists a $j \in J$ such that $e_j$ evaluates to the number zero, then the result of `AND` may be zero (false). Rule (and-true) says that if all $e_j$ for $j \in J$ may evaluate to a non-zero number value, then the result of `AND` may be one (true). In either case, the total cost is the sum of the costs of the subset $J$ of expressions evaluated, plus one. The rules for `OR` are omitted as they are analogous, although dual, to those of `AND`.

Rule (sum) says that if all its argument expressions evaluate to number values then the function call may evaluate to the sum of those values. In Funcalc, functions such as `SUM` and `AVERAGE` are actually more general, and accept a combination of numbers and array values.

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2}{\sigma, \alpha \vdash e_1 = e_2 \Downarrow v_1 = v_2, 1 + c_1 + c_2} \text{ (equal)}$$

$$\frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J. \sigma, \alpha \vdash e_j \Downarrow v_j, c_j \quad \forall j \in J. v_j \in Number \quad \exists j \in J. v_j = 0 \end{array}}{\sigma, \alpha \vdash \texttt{AND}(e_1, \ldots, e_n) \Downarrow 0, 1 + \sum_{j \in J} c_j} \text{ (and-false)}$$

$$\frac{\begin{array}{c} J = \{1, \ldots, n\} \\ \forall j \in J. \sigma, \alpha \vdash e_j \Downarrow v_j, c_j \quad \forall j \in J. v_j \in Number \wedge v_j \neq 0 \end{array}}{\sigma, \alpha \vdash \texttt{AND}(e_1, \ldots, e_n) \Downarrow 1, 1 + \sum_{j \in J} c_j} \text{ (and-true)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n}{\sigma, \alpha \vdash \texttt{SUM}(e_1, \ldots, e_n) \Downarrow \sum_{i=1}^{n} v_i, 1 + \sum_{i=1}^{n} c_i} \text{ (sum)}$$

$$\frac{\begin{array}{ccc} & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 & \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 \\ \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & v_2 \in Number \wedge v_2 \geq 0 & v_3 \in Number \wedge v_3 \geq 0 \end{array}}{\sigma, \alpha \vdash \texttt{CONSTARRAY}(e_1, e_2, e_3) \Downarrow ArrVal(\lfloor v_3 \rfloor, \lfloor v_2 \rfloor, [[v_1 \mid i \leq v_3, j \leq v_2]]), 1 + c_1 + c_2 + c_3 + v_3 \cdot v_2} \text{ (const-array)}$$

$$\frac{\sigma, \alpha \vdash e_0 \Downarrow s, c_0 \qquad s \in Number \wedge 1 \leq s < n + 1 \qquad \sigma, \alpha \vdash e_{\lfloor s \rfloor} \Downarrow v_s, c_s}{\sigma, \alpha \vdash \texttt{CHOOSE}(e_0, e_1, \ldots, e_n) \Downarrow v_s, 1 + c_0 + c_s} \text{ (choose)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow Av(w, h, [[v_{ij}]]), c}{\sigma, \alpha \vdash \texttt{COLUMNS}(e) \Downarrow w, 1 + c} \text{ (columns)}$$

$$\frac{\begin{array}{ccc} & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 & \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 \\ \sigma, \alpha \vdash e_1 \Downarrow Av(w, h, [[v_{ij}]]), c_1 & v_2 \in Number \wedge 1 \leq v_2 < h + 1 & v_3 \in Number \wedge 1 \leq v_3 < w + 1 \end{array}}{\sigma, \alpha \vdash \texttt{INDEX}(e_1, e_2, e_3) \Downarrow v_{\lfloor v_3 \rfloor \lfloor v_2 \rfloor}, 1 + c_1 + c_2 + c_3} \text{ (index)}$$

$$\frac{\begin{array}{cccc} \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 & \sigma, \alpha \vdash e_4 \Downarrow v_4, c_4 & \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 & \sigma, \alpha \vdash e_5 \Downarrow v_5, c_5 \\ v_2 \in Number & v_4 \in Number & v_3 \in Number & v_5 \in Number \\ 1 \leq v_2' \leq h & 1 \leq v_4' \leq h & 1 \leq v_3' \leq w & 1 \leq v_5' \leq w \\ \multicolumn{2}{c}{h' = v_4' - v_2' + 1} & \multicolumn{2}{c}{w' = v_5' - v_3' + 1} \\ \multicolumn{2}{c}{\sigma, \alpha \vdash e_1 \Downarrow Av(w, h, [[v_{ij}]]), c_1} & \multicolumn{2}{c}{v_k' = \lfloor v_k \rfloor, k = 2, 3, 4, 5} \\ \multicolumn{4}{c}{r = ArrVal(w', h', [[v_{i+v_3'-1, j+v_2'-1} \mid v_3' \leq i \leq v_5', v_2' \leq j \leq v_4']]) \qquad c_6 = w' \cdot h'} \end{array}}{\sigma, \alpha \vdash \texttt{SLICE}(e_1, e_2, e_3, e_4, e_5) \Downarrow r, 1 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6} \text{ (slice)}$$

**Fig. 13.** (*continued*).

We do not account for this generality here to avoid overcomplicating the rules.

Rule (const-array) says that if expression $e_1$ may evaluate to a value at some cost $c_1$ and expressions $e_2$ and $e_3$ may evaluate to non-negative numbers, then the call may evaluate to an array value of size $v_3 \cdot v_2$ with $v_1$ as the value of each element. The cost reflects that it is only necessary to evaluate $e_1$ once.

Rule (choose) states that if $e_0$ may evaluate to a number $s \in [1, n+1[$, and $i = \lfloor s \rfloor$, and $e_i$ may evaluate to value $v_i$ at cost $c_i$, then the call may evaluate to $v_i$ at cost $1 + c_0 + c_i$. Since `CHOOSE` is non-strict, we require only that $e_0$ and $e_i$ are evaluated. A more general rule, permitting also speculative evaluation of some $e_j$, could adopt the same approach as for `AND` and have $J = \{0, i\}$ as a special case.

Rule (columns) states that if $e$ may evaluate to an array value then a call to `COLUMNS` may evaluate to the width of that array value. Notice that the cost is pessimistic; an implementation might be able to compute the width of an array value without first fully evaluating it, e.g. if the array value stems from a cell area.

Rule (index) states that if $e_1$ may evaluate to an array value and $e_2$ and $e_3$ may evaluate to numbers within the bounds of the array value, then `INDEX` may evaluate to the value at index $(\lfloor v_3 \rfloor, \lfloor v_2 \rfloor)$. Like rule (columns), the cost is pessimistic.

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \in \mathit{Error}}{\sigma, \alpha \vdash \texttt{ISERROR}(e) \Downarrow 1, 1 + c} \text{ (iserror-true)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \notin \mathit{Error}}{\sigma, \alpha \vdash \texttt{ISERROR}(e) \Downarrow 0, 1 + c} \text{ (iserror-false)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n}{\sigma, \alpha \vdash \texttt{MAX}(e_1, \ldots, e_n) \Downarrow \mathit{max}(v_1, \ldots, v_n), 1 + \sum_{j=1}^{n} c_j} \text{ (max)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow \mathit{Av}(w, h, [[v_{ij}]]), c}{\sigma, \alpha \vdash \texttt{TRANSPOSE}(e) \Downarrow \mathit{Av}(h, w, [[v_{ji} \mid i \leq w, j \leq h]]), 1 + c + w \cdot h} \text{ (transpose)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n}{\sigma, \alpha \vdash \texttt{AVERAGE}(e_1, \ldots, e_n) \Downarrow \frac{1}{n} \sum_{i=1}^{n} v_i, 1 + \sum_{i=1}^{n} c_i} \text{ (average)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n}{\sigma, \alpha \vdash \texttt{HARRAY}(e_1, \ldots, e_n) \Downarrow \mathit{Av}(n, 1, [[v_1, \ldots, v_n]]), 1 + \sum_{i=1}^{n} c_i + n} \text{ (harray)}$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ w = \sum_{i=1}^{n} \mathit{width}(v_i) \quad \forall i, j \in \{1, \ldots, n\}. \mathit{height}(v_i) = \mathit{height}(v_j)\end{array}}{\sigma, \alpha \vdash \texttt{HCAT}(e_1, \ldots, e_n) \Downarrow \mathit{Av}(w, \mathit{height}(v_1), [[v_1 : v_2 : \ldots : v_n]]), 1 + \sum_{i=1}^{n} c_i + n} \text{ (hcat)}$$

**Fig. 13.** (*continued*).

In rule (slice), the premises state that $e_1$ may evaluate to an array value, expressions $e_2$ and $e_4$ may evaluate to a start- and end-column indices, and expressions $e_3$ and $e_5$ may evaluate to a start- and end-row indices, where the indices delimit a sub-array within the input array. Then `SLICE` may evaluate to an array value that is a slice of the original array. The sub-array's size is computed from the row and column indices and its elements are the values of the original array value. The work is one plus evaluating the four indices plus the work of evaluating the input array value plus the size of the new array. The cost is pessimistic; an implementation may return a view of the given (immutable) array value, in which case subcost $c_6$ would be one or even zero.

Rule (iserror-true) states that if $e$ may evaluate to a value $v \in \mathit{Error}$ then the call may evaluate to 1. Rule (iserror-false) is complementary and handles the case where $v \notin \mathit{Error}$.

Rule (max) states that if all the argument expressions may evaluate to numbers at some corresponding costs, then the call may evaluate to the maximum of those values. The rule for `MIN` is analogous.

Rule (transpose) states that if the argument expression may evaluate to an array value of size $w \cdot h$ with cost $c$, then the call may evaluate to a transposed array value of size $h \cdot w$. Notice that element access has been swapped to $v_{ji}$ in the resulting array. The cost is one plus the cost $c$ and the size of the resultant array. This cost is pessimistic; an implementation may represent array values in such a way that `TRANSPOSE` just flips an index-order bit.

Rule (average) is similar to rule (sum) but instead evaluates to the average of the input values. It produces an error if there are no input values (rule not shown).

Rule (harray) states that if the arguments may evaluate to array values $v_i$ with associated costs $c_i$, then the call may evaluate to a single-row array of those values. For example, the expression `=HARRAY(1, HARRAY(2, 3))` will produce an array value of width 2 and height 1 where the first element is the value 1 and the second element is an array with values 2 and 3. The rule for `VARRAY` is similar and has been omitted. The $n$ in the cost of the conclusion denotes the cost of allocating the new array.

Rule (hcat) is closely related to the rule for `HARRAY` but concatenates its arguments, which is why there are additional premises to ensure compatible dimensions (equal heights) of the argument values. The other premises state that the expressions may evaluate to values at some associated costs as in rule (harray). The width of the new array value is the sum of the widths of all its arguments. The function $\mathit{width}$ is defined as follows; function $\mathit{height}$ is analogous: $\mathit{width}(v) = \begin{cases} w & \text{if } v = \mathit{Av}(w, h, [[v_{ij}]]) \\ 1 & \text{otherwise} \end{cases}$

The $n$ cost in the conclusion denotes the cost of concatenation and assumes an efficient implementation of array concatenation. The rule for `VCAT` is similar and has been omitted.

### 4.2. Rules for higher-order intrinsic functions

Since most higher-order functions call some supplied function multiple times, we introduce quantification over the environment $\rho'$ used to evaluate a sheet-defined function. For example, `TABULATE` calls the supplied function $f(i, j)$ with the row and column indices $(i, j)$ for each position of an array passed to `TABULATE`. E.g. `TABULATE`$(f, 3, 4)$ with $f(i, j) = i * 10 + j$, would produce the table

| | | | |
|---|---|---|---|
| 11 | 21 | 31 | 41 |
| 12 | 22 | 32 | 42 |
| 13 | 23 | 33 | 43 |

We quantify over $\rho'$ with the current position $(i, j)$ as $\rho'_{ij}$ to have a separate fresh environment, akin to a separate stack frame, for each call $f(i, j)$. Similarly, we also quantify over the function call cost environment $\gamma'$ as $\gamma'_{ij}$. We start by introducing the rule for TABULATE in full detail, then define auxiliary notation so that we do not need to repeat ourselves in the remaining rules.

$$\frac{\begin{array}{c}\sigma,\alpha\vdash e_1\Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_1\\ def(sdf)=(out,[in_1,\ldots,in_{k+2}],cells)\\ \sigma,\alpha\vdash e_2\Downarrow v_2,c_2\quad v_2\in Number\wedge h=\lfloor v_2\rfloor\geq 0\\ \sigma,\alpha\vdash e_3\Downarrow v_3,c_3\quad v_3\in Number\wedge w=\lfloor v_3\rfloor\geq 0\\ \forall i,j.\,\rho'_{ij},\gamma'_{ij}\ fresh\quad \forall i,j.\,\rho'_{ij}(in_1)=u_1\ \ldots\ \rho'_{ij}(in_k)=u_k\\ \forall i,j.\,\rho'_{ij}(in_{k+1})=i\wedge\rho'_{ij}(in_{k+2})=j\\ \forall i,j.\,\forall ca\in dom(\rho'_{ij})\setminus\{in_1,\ldots,in_{k+2}\}.\,\sigma,\alpha\vdash\phi(ca)\Downarrow\rho'_{ij}(ca),\gamma'_{ij}(ca)\\ \forall i,j.\,v_{ij}=\rho'_{ij}(out)\quad c_4=\sum_{i,j}\sum_{ca\in dom(\gamma'_{ij})}\gamma'_{ij}(ca)\end{array}}{\sigma,\alpha\vdash\texttt{TABULATE}(e_1,e_2,e_3)\Downarrow Av(w,h,[[v_{ij}]]),1+c_1+c_2+c_3+c_4+w\cdot h}\ \text{(tabulate-full)}$$

Taking the premises in order from top to bottom, they state that $e_1$ may evaluate to a function value, at cost $c_1$, that expects two more arguments, and that $e_2$ and $e_3$ may evaluate to non-negative numbers of $h$ rows and $w$ columns respectively, after truncation towards zero. We then postulate $w \cdot h$ fresh environments $\rho'_{ij}$ where $i \leq w$ and $j \leq h$, one for each application of the function value. The next quantified premises state that the input cells should contain the early-bound values $[u_1, \ldots, u_k]$ except the last two arguments which must be the indices $i$ and $j$ of the function application. The following quantified premise states that for all cell addresses $ca$ in the domain of $\rho'_{ij}$, excluding the set of input cells, the expression of each cell address may evaluate to the value given by environment $\rho'_{ij}$ at some cost. The final premise states that each function application evaluates to the value $v_{ij}$ of the function call's output cell. The call to TABULATE then evaluates to an array value of size $w \cdot h$ whose elements are the $v_{ij}$. The work is 1 plus the costs for evaluating the function value, the two arguments denoting the desired size of the array, the sum of the costs of applying the function for every position $(i, j)$, and allocating a new array.

The premises that pass values to the function arguments, evaluate the cells of the function's body, get the result from the function's output cell and compute the total cost of evaluating the function are of a more general nature: this is how all higher-order functions call function values. So to reduce repetition, we define a predicate *apply* as follows:

$$apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],a_1,\ldots,a_n,r,c)\triangleq\begin{cases}def(sdf)=(out,[in_1,\ldots,in_{k+n}],cells)\\ \rho'(in_1)=u_1\ \ldots\ \rho'(in_k)=u_k\quad\rho'(in_{k+1})=a_1\ \ldots\ \rho'(in_{k+n})=a_n\\ \forall ca\in dom(\rho')\setminus\{in_1,\ldots,in_{k+n}\}.\,\rho',\sigma\vdash\phi(ca)\Downarrow\rho'(ca),\gamma'(ca)\\ r=\rho'(out)\\ c=\sum_{ca\in dom(\gamma'_{ij})}\gamma'(ca)\end{cases}$$

Note that the "arguments" $r$ and $c$ are the function call's result and its cost; these are "output parameters" of *apply* in much the same style as when a Prolog predicate is used to represent a function. The definition of *apply* can be used to rewrite the rule for TABULATE for a clearer and more compact rule:

$$\frac{\begin{array}{c}\sigma,\alpha\vdash e_1\Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_1\\ \sigma,\alpha\vdash e_2\Downarrow v_2,c_2\quad v_2\in Number\wedge h=\lfloor v_2\rfloor\geq 0\\ \sigma,\alpha\vdash e_3\Downarrow v_3,c_3\quad v_3\in Number\wedge w=\lfloor v_3\rfloor\geq 0\\ \forall i,j.\,apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],i,j,v_{ij},c_{ij})\qquad c_4=\sum_{i,j}c_{ij}\end{array}}{\sigma,\alpha\vdash\texttt{TABULATE}(e_1,e_2,e_3)\Downarrow Av(w,h,[[v_{ij}]]),1+c_1+c_2+c_3+c_4+w\cdot h}\ \text{(tabulate)}$$

Before moving on to the remaining higher-order functions, we need to address recursion. For some functions, such as REDUCE, we need to handle intermediate computation steps that operate on values that we do not want to compute repeatedly. For this purpose we introduce a new judgment form $\sigma,\alpha\vdash_v s\Downarrow v,c$ that operates on values instead of expressions. The judgment states that given the usual environments $\sigma$ and $\alpha$, some intermediate value-based computation state $s$ may evaluate to a value $v$ at cost $c$. This allows us to evaluate the top-level argument expressions once in the initial call to REDUCE, and then use their values in subsequent recursive calls without having to recompute these expressions.

The REDUCE function takes three arguments: a two-argument function value $f$; an initial value $x_0$; and an array value $av$, and performs a reduction over the elements of $av$ using $f$ and the initial value $x_0$. The call REDUCE(CLOSURE("-"), 0, HCAT(1, 2, 3)) would compute $((0-1)-2)-3=-6$.

$$\frac{\begin{array}{cc}\sigma,\alpha\vdash e_1\Downarrow v_1,c_1 & v_1=FunVal(sdf,[u_1,\ldots,u_k])\\ k=arity(sdf)-2 & \sigma,\alpha\vdash e_3\Downarrow v_3,c_3\\ \sigma,\alpha\vdash e_2\Downarrow v_2,c_2 & v_3\in ArrVal\\ \multicolumn{2}{c}{\sigma,\alpha\vdash_v\texttt{REDUCE}(v_1,v_2,v_3)\Downarrow r,c_r}\end{array}}{\sigma,\alpha\vdash\texttt{REDUCE}(e_1,e_2,e_3)\Downarrow r,1+c_1+c_2+c_3+c_r}\ \text{(reduce)}$$

We start with the top-level, expression-based rule (reduce) for the REDUCE function which passes the results of the evaluated expressions to a value-based rule. This rule in turn arbitrarily decomposes the array value until a single value is left which is then combined with the intermediate result using the function value. Expression $e_1$ may evaluate to a function value, $e_2$ to an initial value for the reduction, and $e_3$ to an array $v_3$. Each value is passed as an argument to the value-based reduction rule defined next.

$$\frac{\begin{array}{cc}v_3=v_l:v_r & v_l\in ArrVal\wedge v_r\in ArrVal\\ v_1\in FunVal & \sigma,\alpha\vdash_v\texttt{REDUCE}(v_1,v_2,v_l)\Downarrow r_l,c_l\\ v_2\in Number & \sigma,\alpha\vdash_v\texttt{REDUCE}(v_1,r_l,v_r)\Downarrow r,c_r\end{array}}{\sigma,\alpha\vdash_v\texttt{REDUCE}(v_1,v_2,v_3)\Downarrow r,1+c_3+c_l+c_r}\ \text{(reduce-inductive)}$$

The inductive value-based reduction rule (reduce-inductive) first decomposes $v_3$ into two arrays $v_l$ and $v_r$. This can be an arbitrary decomposition as chosen by an implementation since, given an identity element and an associative binary function, a reduction may e.g. proceed from left to right using a decomposition similar to functional lists; or decompose the operations as a tree by recursively splitting the input array in halves to perform the reduction in parallel. Notice that we pass the result of the reduction of the left part $v_l$ of the decomposed array as the initial value of the reduction of the right part $v_r$ of the decomposed array. The reason is purely semantic and will become apparent shortly.

We need two additional base case rules to account for a reduction of a single value and for an empty array value. These are given as rules (reduce-base-singular) and (reduce-base-empty).

$$\frac{\begin{array}{c}v_1=FunVal(sdf,[u_1,\ldots,u_k])\\ v_3=Av(1,1,[[v_{11}]])\\ apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],v_2,v_{11},r,c)\end{array}}{\sigma,\alpha\vdash_v\texttt{REDUCE}(v_1,v_2,v_3)\Downarrow r,1+c}\ \text{(reduce-base-singular)}$$

$$\frac{v_1\in FunVal\qquad v_2\in Number\qquad v_3=Av(0,0,[[]])}{\sigma,\alpha\vdash_v\texttt{REDUCE}(v_1,v_2,v_3)\Downarrow v_2,1}\ \text{(reduce-base-empty)}$$

Rule (reduce-base-singular) handles the case where $v_3$ is a single-element array and we pass the single element $v_{11}$ and starting value $v_2$ to the sheet-defined function $sdf$ from the function value $v_1$. In rule (reduce-inductive), the accumulated intermediate results $v_2$ and $r_l$ must be threaded through the subcomputations. Rule (reduce-base-empty) returns the starting value $v_2$ of the reduction if passed the empty array.

$$\dfrac{\dfrac{\dfrac{apply_{\sigma,\alpha}(sdf,[],8,1,r,c_a)}{\sigma,\alpha\vdash_v \texttt{REDUCE}(Fv(sdf,[]),8,[1])\Downarrow 9,1+c_a} \quad \dfrac{apply_{\sigma,\alpha}(sdf,[],9,2,r,c_a)}{\sigma,\alpha\vdash_v \texttt{REDUCE}(Fv(sdf,[]),9,[2])\Downarrow 11,1+c_a} \quad [1,2]=[1]:[2]}{\sigma,\alpha\vdash_v \texttt{REDUCE}(Fv(sdf,[]),8,[1,2])\Downarrow 11,1+2+2c_a} \quad \dfrac{\dfrac{apply_{\sigma,\alpha}(sdf,[],11,3,r,c_a)}{\sigma,\alpha\vdash_v \texttt{REDUCE}(Fv(sdf,[]),11,[3])\Downarrow 14,1+c_a} \quad \dfrac{apply_{\sigma,\alpha}(sdf,[],14,4,r,c_a)}{\sigma,\alpha\vdash_v \texttt{REDUCE}(Fv(sdf,[]),14,[4])\Downarrow 18,1+c_a} \quad [3,4]=[3]:[4]}{\sigma,\alpha\vdash_v \texttt{REDUCE}(Fv(sdf,[]),11,[3,4])\Downarrow 18,1+2+c_d+2c_a} \quad [1,2,3,4]=[1,2]:[3,4]}{\sigma,\alpha\vdash_v \texttt{REDUCE}(Fv(sdf,[]),8,[1,2,3,4])\Downarrow 18,1+3+3+4c_a} \quad \sigma,\alpha\vdash \texttt{CLOSURE("+")}\Downarrow Fv(sdf,[]),c_1 \quad \sigma,\alpha\vdash 8\Downarrow 8,1 \quad \sigma,\alpha\vdash \texttt{HCAT(1, 2, 3, 4)}\Downarrow [1,2,3,4],c_h$$
$$\sigma,\alpha\vdash \texttt{REDUCE(CLOSURE("+"), 8, HCAT(1, 2, 3, 4))}\Downarrow 18,1+7+4c_a+c_h+c_1$$

**Fig. 14.** The full derivation tree of the expression `=REDUCE(CLOSURE("+"), 8, HCAT(1, 2, 3, 4))` using a combination of the expression- and value-based rules for reduction.

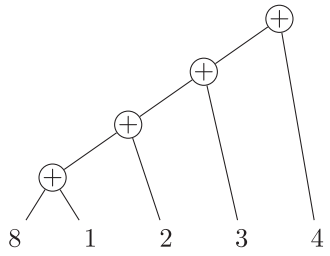


**Fig. 15.** The tree for the reduction in the REDUCE example.

To illustrate these rules, we expand the derivation tree for the following expression in Fig. 14 and show the corresponding tree decomposition in Fig. 15.

```
= REDUCE(CLOSURE("+"), 8, HCAT(1, 2, 3, 4))
```

In the example, $c_a$ is the cost of applying the addition operator, and $c_h$ is the cost of concatenating elements with HCAT. Additionally, we have omitted some of the set membership tests for values to keep the derivation tree succinct. For brevity, we denote lists in square braces $[1,2,3,4]$. The result of the expression is $8+1+2+3+4=18$.

We are now ready to list the rules for the remaining higher-order functions in Funcalc which are shown in Fig. 16.

Rule (map) shows the rule for the MAP function which is in fact an $n$-ary zip-by function, with the ordinary MAP function as the special case $n=1$. In general, if $e_0$ evaluates to a function value, and the other arguments $e_1,\ldots,e_n$ evaluate to array values of equal size, then for each array position $(i,j)$ the function is applied to the $n$ arguments $v^1_{ij},\ldots,v^n_{ij}$ giving $r_{ij}$ and the final result is the array of these $r_{ij}$. The total cost is one plus the cost of evaluating the function value, the cost of evaluating all the array value arguments, and the cost of all function applications.

Rule (colmap) states that if the first argument may evaluate to a function $f$ and the second argument may evaluate to an array value whose height equals $f$'s arity, then the call may evaluate to a new single-row array value whose elements are the results of applying $f$ to each column in the input array. A similar function ROWMAP exists for row-wise mapping to a single-column array value.

Rule (countif) states that if the first argument $e_0$ evaluates to a unary predicate function $f$ and the remaining expressions to some values, then a call to COUNTIF may evaluate to the number of those values for which $f$ returned a non-zero number (true). The total cost is one plus evaluating the function expression, all argument expressions and the cost of each function application.

Finally, we have the rule for HSCAN. The function performs a column-wise inclusive scan operation as opposed to an element-wise scan as per Blelloch [8]. For example, given function $f(x)=x+1$ we might call HSCAN as an array formula in the cell range A4:C5 on the cell range A1:A2 as shown in Fig. 17.

```
HSCAN(CLOSURE("f"), A1 : A2, 2)
```

Except the starting column, the values in each column of the result are one greater than the corresponding values in the preceding column.

Rule (hscan) says that if $e_1$ evaluates to a function value accepting one argument, and $e_2$ evaluates to a column array value $r_0$, and $e_3$ evaluates to a number $n$, the result will be an array $arr$ with $n+1$ columns, column number $i$ being the result of $f^i(r_0)$. The total cost is one plus the costs of evaluating the arguments plus the cost $n$ of the column concatenations plus the cost of the function applications.

## 5. Abstract cost semantics

Gomez et al. and Rosendahl worked on cost translations for higher-order functional languages [12,13] which cater for computing with unknown data values. Adding such unknown values allows for a rudimentary abstract interpretation of programs which in many cases can provide a rather precise approximation of the actual cost of the computation.

However, there is no standard way to insert an unknown value in a sheet. If a cell is empty and a lookup is performed on the cell, the value 0.0 is returned as is consistent with the standard semantics. However, it may be interesting to abstractly evaluate sheets where some cells have unknown values. This can be handled by a built-in function UNKNOWN taking no arguments and always returning the unknown value, denoted $\top$, which is the top-most abstract value in an abstract lattice. Calling UNKNOWN takes one computational step.

To allow computations with unknown values, we need an abstract representation of values that models the concrete values used in concrete interpretation or more generally we need a domain, which is a set equipped with a partial order, denoted $\sqsubseteq$. The domain has a least element, denoted $\bot$ and if the domain is a lattice, it also has a top element, denoted $\top$. A domain is also equipped with a join operation, denoted $\sqcup$ [27]. Fig. 18 gives a suggested lattice of abstract values $AbsValue$ for Funcalc which can be used in the definition of abstract values computed by an abstract semantics for Funcalc. The sets and maps used for abstract interpretation are shown in Fig. 19 and the abstract cost semantics rules are depicted in Fig. 20.

The ordering on $AbsValue$ is such that $\forall absv \in AbsValue\ .\ absv \sqsubseteq \top$. Furthermore, $\forall absav \in AbsArrVal\ .\ absav \sqsubseteq Array(\top,\top)$ where $AbsArrVal$ denotes the set of all possible abstract array values. We can now follow the ideas presented by Schmidt [27] and provide a trace-based abstract interpretation for Funcalc, based on the ideas for big step semantics presented in section 5 of [27].

The cost semantics for Funcalc presented in Section 3 is extended with the following rules:

Rule (u1a) states that calling the built-in function UNKNOWN, taking no arguments, returns the unknown value $\top$ with a cost of one computational step.

Rule (g3a) states that if the predicate, i.e. the first argument to IF, evaluates to the unknown value $\top$ at some cost $c_1$, then the resulting value is the join of the values, denoted by $\sqcup$, of the evaluation of the two branches, and the total cost of evaluating IF is the cost $c_1$ plus the maximum of the cost of evaluating either of the two branches.

Rule (g5a) states that if any argument to a built-in function evaluates to the unknown value $\top$, then the resulting value is the unknown value $\top$, i.e. all built-in functions are strict wrt. the unknown value $\top$. However, the cost of calling a built-in function with the unknown value $\top$ is still the cost of evaluating the arguments, plus the cost of the function call and the cost of the work of the built-in function. This is a bit conservative, but allows the implementation of built-in functions to do some work before returning the unknown value $\top$.

Rule (g10a) states that if the function in an APPLY, i.e. the first argument, evaluates to the unknown value $\top$ then the resulting value

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_1 \Downarrow Av(w_1,h_1,[[v^1_{ij}]]),c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow Av(w_n,h_n,[[v^n_{ij}]]),c_n \\ \sigma,\alpha \vdash e_0 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_0 \qquad \forall l,m \in \{1,\ldots,n\}.\, w_l = w_m \wedge h_l = h_m \\ k = arity(sdf) - n \qquad \forall i \in \{1,\ldots,w_1\}, j \in \{1,\ldots h_1\}.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],v^1_{ij},\ldots,v^n_{ij},r_{ij},c_{ij})\end{array}}{\sigma,\alpha \vdash \texttt{MAP}(e_0,e_1,\ldots,e_n) \Downarrow Av(w_1,h_1,[[r_{ij}]]),1+c_0+\sum_{l=1}^n c_l+\sum_{ij} c_{ij}} \text{(map)}$$

$$\frac{\begin{array}{cc}\sigma,\alpha \vdash e_1 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_1 & \sigma,\alpha \vdash e_2 \Downarrow Av(w,h,[[v_{ij}]]),c_2 \\ k = arity(sdf) - h & \forall i \in \{1,\ldots,w\}.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],[[v_{i*}]],r_i,c_i)\end{array}}{\sigma,\alpha \vdash \texttt{COLMAP}(e_1,e_2) \Downarrow Av(w,1,[[r_i]]),1+c_1+c_2+\sum_i c_i} \text{(colmap)}$$

$$\frac{\begin{array}{cc}\sigma,\alpha \vdash e_0 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_0 & \sigma,\alpha \vdash e_1 \Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow v_n,c_n \\ k = arity(sdf) - 1 & \forall i \in \{1,\ldots,n\}.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],v_i,r_i,t_i)\end{array}}{\sigma,\alpha \vdash \texttt{COUNTIF}(e_0,e_1,\ldots,e_n) \Downarrow \sum_{i=1}^n 1_{r_i \neq 0},1+c_0+\sum_{j=1}^n c_j+\sum_{i=1}^n t_i} \text{(countif)}$$

$$\frac{\begin{array}{cc}\sigma,\alpha \vdash e_1 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_1 & \sigma,\alpha \vdash e_3 \Downarrow v_3,c_3 \\ k = arity(sdf) - 1 & v_3 \in Number \\ \sigma,\alpha \vdash e_2 \Downarrow Av(1,h,[[v_{1*}]]),c_2 & n = \lfloor v_3 \rfloor \\ r_0 = [[v_{1*}]] & \forall i \in \{1,\ldots,n\}.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],r_{i-1},r_i,t_i)\end{array}}{\sigma,\alpha \vdash \texttt{HSCAN}(e_1,e_2,e_3) \Downarrow Av(n,h,[[r_0:\ldots:r_n]]),1+c_1+c_2+c_3+c_c \cdot n+\sum_{i=1}^n t_i} \text{(hscan)}$$

**Fig. 16.** Operational and cost semantics for a subset of Funcalc's higher-order built-in functions. For very similar rules such as for HCAT and VCAT, we omit repetitions and give just one of them.

| | A | B | C |
|---|---|---|---|
| 1 | 1 | | |
| 2 | 2 | | |
| 3 | | | |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 4 |

**Fig. 17.** Example column-wise scan using HSCAN with input range A1:A2 and results in cells A4:C5, where each column's values are the preceding column's values, plus one.

is the unknown value $\top$, and the cost is the top element in the cost domain, i.e. $\infty$. The standard semantics for APPLY does not state an evaluation order, but APPLY is strict in all its arguments. An alternative rule for APPLY could be stated as follows:

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_0 \Downarrow \top,c_0 \\ \sigma,\alpha \vdash e_1 \Downarrow v_1,c_1 \qquad \ldots \qquad \sigma,\alpha \vdash e_n \Downarrow v_n,c_n\end{array}}{\sigma,\alpha \vdash \texttt{APPLY}(e_0,e_1,\ldots,e_n) \Downarrow \top,\infty} (g10alt)$$

Rule (g10alt) would enforce strict evaluation of all arguments to APPLY, thereby excluding an implementation using short-cut semantics for the first argument. Since a short-cut semantics is a quite natural implementation strategy, we have chosen to use the rule (g10a) instead of rule (g10alt) as the resulting cost is the same, namely $\infty$.

With these rules, it is possible to establish a safety property for finite derivations [27]. First we define a safety property for values and costs:

$v,c$ safe$_{val}$ $av,ac$ iff $v \sqsubseteq av$ and $c \sqsubseteq ac$

Here we use the ordering on values defined above and the usual order on integer costs augmented with $\forall c\ .\ c \leq \infty$. The property states that value $v$ and cost $c$ are safely approximated by abstract value $av$ and abstract cost $ac$ if and only if $av$ is ordered above $v$ in the abstract value lattice and $ac$ is ordered above $c$, respectively. As mentioned in [28] this relation is a Galois connection between the concrete and the abstract domains.

The safety property on values is then extended to environments:

$$\begin{aligned}\sigma,\alpha \text{ safe}_{env}\ \sigma_a,\alpha_a \text{ iff } &\texttt{dom}(\sigma) = \texttt{dom}(\sigma_a) \\ &\wedge\ \texttt{dom}(\alpha) = \texttt{dom}(\alpha_a) \\ &\wedge \forall ca\ .\ \sigma(ca) \text{ safe}_{val}\ \sigma_a(ca) \\ &\wedge \forall ae\ .\ \alpha(ae) \text{ safe}_{val}\ \alpha_a(ae)\end{aligned}$$

Finally the safety property can be extended to judgments.

$$\begin{aligned}&\sigma,\alpha \vdash e \Downarrow_t v,c \text{ safe}_{seq}\ \sigma_a,\alpha_a \vdash e \Downarrow_{at} av,ac \\ &\quad \texttt{iff}\ \sigma,\alpha \text{ safe}_{env}\ \sigma_a,\alpha_a \text{ and } v,c \text{ safe}_{val}\ av,ac\end{aligned}$$

where $\Downarrow_t$ indicates that the transition $\Downarrow$ is established using the rules from Section 3 and $\Downarrow_{at}$ indicate that transition $\Downarrow$ is established also using the additional abstract rules presented above.

With the safety property on judgments we can extend the definition to trees safe$_{tree}$. For trees $T_C$ resp. $T_A$ the proposition $T_C$ safe$_{tree}$ $T_A$ holds if $root(T_C)$ safe$_{seq}$ $root(T_A)$ and for every child subtree $t_i$ of $T_C$ there exists a subtree $t_j$ of $T_A$ such that $t_i$ safe$_{tree}$ $t_j$ holds.

Let wftree$_C$ and wftree$_A$ be the set of well formed proof trees in the concrete semantics, respectively the set of well formed proof trees in the abstract semantics. For every expression $e$ and concrete environments $\sigma,\alpha$ and abstract environments $\sigma_a,\alpha_a$, where $\sigma,\alpha$ safe$_{env}$ $\sigma_a,\alpha_a$ holds, we can establish the desired property that for every proof tree $t_C \in$ wftree$_C$, where root$(t_C) = \sigma,\alpha \vdash e \Downarrow_t v,c$, and for every proof tree $t_A \in$ wftree$_A$, where root$(t_A) = \sigma_a,\alpha_a \vdash e \Downarrow_{at} av,ac$, it is the case that $t_C$ safe$_{tree}$ $t_A$. The proof of this follows by induction on the height of the derivation tree.

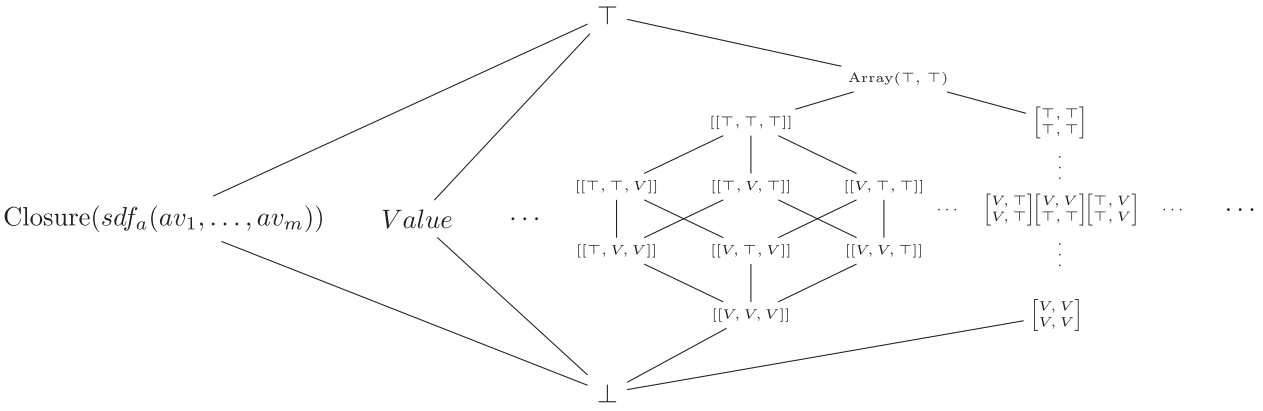


**Fig. 18.** A lattice of abstract values that can be used in abstract interpretation of Funcalc. The symbol ⊤ denotes that a value can be represented by multiple values in the lattice, e.g. something is both an atomic value and an array of known size. In type systems, this constitutes a type unification error. The symbol ⊥ denotes that we know nothing about a value. The Values are either the semantic map of numbers and errors, abstract closures, or array values. For arrays, we need both an abstraction for an array of unknown size Array(⊤, ⊤) and arrays of known sizes.

$$
\begin{array}{llll}
n & \in & Number & = \{ \text{ IEEE floating} - \text{point numbers } \} \\
absav & \in & AbsArrVal & = \{ (w, h, [[absv_{ij} \mid i \leq w, j \leq h]]) \} + \{(\top, \top)\} \\
absfv & \in & AbsFunVal & = \{ (sdf, [absu_1, \ldots, absu_k]) \} \\
 & & Error & = \{ \texttt{\#DIV/0!}, \texttt{\#CYCLE!}, \texttt{\#NA} \} \\
ca & \in & Addr & = \{ \text{ cell addresses } (c, r) \} \\
absv, absu & \in & AbsValue & = Number + Error + AbsArrVal + AbsFunVal + \{\top\} \\
e & \in & Expr & = \{ \text{ formulas, see Figure 6 } \} \\
\phi_a & & & \in Addr \to Expr \\
\sigma_a & & & \in Addr \to AbsValue \\
\alpha_a & & & \in Expr \to AbsValue \\
\rho_a & & & \in Addr \to AbsValue
\end{array}
$$

**Fig. 19.** Sets and maps used in the abstract Funcalc semantics. The tuple $\{(\top, \top)\}$ component of *AbsArrVal* represents an array of unknown width and height.

$$\frac{}{\sigma, \alpha \vdash \texttt{UNKNOWN}() \Downarrow \top, 1} \; (u1a)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow \top, c_1 \qquad \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 \qquad \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3}{\sigma, \alpha \vdash \texttt{IF}(e_1, e_2, e_3) \Downarrow v_2 \sqcup v_3, 1 + c_1 + max(c_2, c_3)} \; (g3a)$$

$$\frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\ \sigma, \alpha \vdash e_j \Downarrow v_j, c_j \qquad v_i = \top \text{ for some } i \in J \end{array}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow \top, 1 + \sum_{j \in J} c_j + work(f, v_1, \ldots, v_n)} \; (g5a)$$

$$\frac{\sigma, \alpha \vdash e_0 \Downarrow \top, c_0}{\sigma, \alpha \vdash \texttt{APPLY}(e_0, e_1, \ldots, e_n) \Downarrow \top, \infty} \; (g10a)$$

**Fig. 20.** Abstract cost (or work) semantics rules.

The only non-trivial base case is when rule (u1a) has been applied. Cells filled with a call to the built-in function UNKNOWN are filled with a constant value "in a production sheet". Any constant value is also an element of *AbsValue* and since the ordering on *AbsValue* is such that $\forall absv \in AbsValue\ .\ absv \sqsubseteq \top$ this base case holds. Of the three inductive cases (g3a), (g5a) and (g10a), rule (g3a), the rule for $IF(e_1, e_2, e_3)$, is the most interesting. This rule is only applicable if the condition $e_1$ cannot be evaluated to a concrete value in the abstract cost semantics, i.e. $e_1$ evaluates to ⊤. If the concrete value evaluates to 0.0 in the concrete semantics, the rule (g3f) will be applied in the concrete semantics yielding a value $v_3$ and a cost $c = 1 + c_1 + c_3$ where $c_1$ is the cost

of evaluating the condition $e_1$, and $c_3$ is the cost of evaluating the false-branch $e_3$. Clearly $v_3 \sqsubseteq v_2 \sqcup v_3$ and $1+c_1+c_3 \sqsubseteq 1+c_1+max(c_2, c_3)$, where $v_2$ is the value of evaluating the true-branch and $c_2$ is the associated cost. The case when $e_1$ evaluates to a value different from 0.0 is similar.

The above only establishes a safety property for finite derivations, i.e. for terminating programs. Not all programs terminate and future work needs to look into handling infinite derivations as well, possibly following ideas presented in [27].

## 6. Implementation of cost semantics

Before we discuss the full implementation details of the various sections presented thus far, we give a brief introduction to some of the inner workings of the research spreadsheet application Funcalc deemed necessary for understanding the implementation. Readers interested in learning more are encouraged to read [7].

### 6.1. Funcalc

While sheet-defined functions are compiled to Common Intermediate Language (CIL) bytecode, the expressions of ordinary cells are evaluated to values by an interpreter. To express different types of cells and expressions, Funcalc uses class hierarchies with base class `Cell` for cells and base class `Expr` for expressions. For example, in the case of cells, `NumberCell` and `TextCell` contain number and text constants respectively while `Formula` holds a formula expression such as `=1+2`. Expressions may e.g. be a constant `Const`, cell reference `CellRef`, or a function call `FunCall`. Similarly, there is a class hierarchy for values with base class `Value`. For example, the `ArrayValue` class represents a first-class array value.

Funcalc's interpreter implements two interfaces for evaluating cells (`ICellEvaluator`) and expressions (`IExpressionEvaluator`), the former interface is given in Listing 1 with some details omitted. Evaluation of a cell happens in the context of some column and row in some sheet. These interfaces can be implemented by any class that needs to operate on cells, expressions or both, and are used to implement the rules of our cost semantics.

```
1  public interface ICellEvaluator<T>
2  {
3      T Eval(ArrayFormula cell, Sheet sheet, int col,
           int row);
4      T Eval(BlankCell cell,    Sheet sheet, int col,
           int row);
5      T Eval(Formula cell,      Sheet sheet, int col,
           int row);
6      T Eval(NumberCell cell,   Sheet sheet, int col,
           int row);
7      T Eval(TextCell cell,     Sheet sheet, int col,
           int row);
8  }
```

**Listing 1:** The interface for evaluating cells in Funcalc.

Instead of returning a value like the standard interpreter, the cost interpreter returns a `CostResult` consisting of both the `Value` and its cost. We define some auxiliary functions like `MakeCostResult` which constructs a `CostResult` tuple from a pre-existing `CostResult` or from a value and a cost. These auxiliary functions ensure that cost results are monotonically increasing by adding a unit cost of 1.

We have implemented two variants of the cost semantics in Funcalc: a concrete cost evaluator (Section 6.2) and an abstract cost interpreter (Section 6.4). The former uses unit costs and is not guaranteed to terminate e.g. in the presence of infinite recursion, the latter is inspired by [27,29,30]. We also discuss a few important details regarding cost evaluation of sheet-defined functions.

### 6.2. Cost evaluator implementation

The implementation of the cost evaluator follows the semantic cost rules closely as shown in Listing 2 for the simplified implementation of the cost evaluation of `IF` (see rules $(c3e)$, $(c3f)$ and $(c3t)$ in Section 2.1). The evaluation function `EvalIf` takes the function call expression `FunCall` representing the `IF` expression and the column, row and sheet of the cell. First, we check if the function call consists of three sub-expressions (a condition and two branches). If not, we return an error indicating an incorrect number of arguments. Otherwise, we evaluate the conditional expression. If the result is an error value, we short-circuit as per rule $(c3e)$ and return the result of the condition (the error) and the cost obtained so far. Otherwise, we cast the result of the condition to a number. If the cast fails, we return an error indicating an argument type error and the cost obtained thus far. If the condition is indeed a number, we pick the appropriate branch and evaluate the expression as per rules $(c3f)$ or $(c3t)$, then return its value along with the cost of evaluating the condition and the given branch expression. As an example, `EvalIf` would return a cost result consisting of the value `SIN(1+2)`$\approx 0.14112$ at cost 6 for the following expression:

```
=IF(1, SIN(1+2), COS(3))
```

Evaluation of the `IF` function call and its condition costs 2 units. The inner function call to `SIN` costs four units: one for the `SIN` function application, one for the `+` operator application and one for each of the arguments of the addition.

```
1  private CostResult EvalIf(FunCall expr, Sheet sheet,
       int col, int row)
2  {
3      if (expr.expressions.Length != 3) {
4          return MakeUnitCost(ErrorValue.argCountError)
               ;
5      }
6
7      CostResult condition = expr.es[0].Eval(this,
           sheet, col, row);
8
9      if (condition.Value is ErrorValue ev) {
10         return MakeCostResult(condition);
11     } else {
12         NumberValue n0 = condition.Value as
               NumberValue;
13
14         if (n0 != null) {
15             int index = n0.value != 0.0 ? 1 : 2;
16             CostResult result = expr.expressions[
                   index].Eval(this, sheet, col, row);
17
18             return MakeCostResult(result.Value,
                   condition.Cost + result.Cost);
19         } else {
20             return MakeCostResult(ErrorValue.
                   argTypeError, condition.Cost);
21         }
22     }
23 }
```

**Listing 2:** Simplified C# code for the cost evaluation of `IF`

### 6.3. Evaluation of sheet-defined functions

In Funcalc, sheet-defined functions are not interpreted but automatically compiled to CIL bytecode [7]. Without extending the semantics to incorporate CIL bytecode, we cannot use the existing interpreter framework to find the cost of evaluating sheet-defined functions. We could generate additional code to compute costs but this seems like an excessive and complicated approach. Instead, we directly interpret the cells of a sheet-defined function using the cost evaluator by evaluating

| | A | B |
|---|---|---|
| 1 | =DEFINE("factorial", B3, B2) | |
| 2 | 'n= | 0 |
| 3 | 'out= | =IF(B2<=0, 1, B2*FACTORIAL(B2-1)) |

**Sheet 1.** A recursive factorial sheet-defined function.

the output cell of a sheet-defined function and following dependencies back to its input cells. This requires proper abstraction of $\rho : Addr \to Value$, that is, the local cell environment or stack frame of a sheet-defined function, to handle both recursive sheet-defined functions and normal function calls. Consider the definition of the factorial function in Sheet 1.

To implement $\rho$, we could directly modify the input cells of the sheet-defined function on each call but this would temporarily modify cells in the spreadsheet which could easily lead to inconsistencies if we are not careful. Instead, we keep track of an internal, local environment $lenv : Addr \to Value$ that mimics $\rho$. 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 (g8) for application, where the input parameters for the current function call are stored in a fresh environment $\rho'$ i.e. $\rho'(in_1) = v_1 \dots \rho'(in_n) = v_n$. When calling a function recursively, 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. Lastly, cost evaluation of a cell reference is modified to first perform a lookup of the address in the top-most local environment, if any, before examining the cells of the actual sheets. Thus when we do computation in some recursive sheet-defined function and need to evaluate an input parameter, we first look in the local environment and not the actual spreadsheet.

The above local environment scheme combined with evaluating the output cell first, ensures that we only evaluate the cells that are necessary for computing the output cell as discussed in Section 3.2. Argument evaluation for intrinsic functions is implemented in a left to right order in the interpreter. This removes the nondeterminism afforded by the error rule (c5e), as evaluation of arguments will be terminated when an argument evaluates to an error, thus fixing $J = \{1, \dots, i\}$ as the least prefix of argument indexes for which $v_i$ is an error.

Interestingly, if we were to strip away any notion of cost from the evaluation of sheet-defined functions, we have in fact implemented a full-fledged sheet-defined function interpreter which is likely what Funcalc would have used if there was no sheet-defined function compiler. One issue with the above approach is that the cost of interpretation and the cost of bytecode execution may not correlate, since the point of compiling sheet-defined functions is that bytecode execution should be much faster than interpretation of expressions. This is not a problem as long as cost is only used as a measure of computational steps. However, if we were interested in worst case execution times, tighter correspondence with the execution time of CIL bytecode becomes paramount.

### 6.4. Abstract cost evaluator implementation

This section presents examples from the abstract cost semantics implementation. The abstract cost implementation introduces a new type of value, `Top` to represent $\top$ values as discussed in Section 5; recall that $\top$ represents unknown values, such as input values for the spreadsheet, and values that through computation depend on $\top$. Essentially, this is just a subclass `Top` of `Value`; a function `UNKNOWN` is introduced to produce a $\top$ value.

The abstract cost implementation is an implementation of the *evaluator* interfaces, and is a modified version of the `CostEvaluator`. Specifically, the difference is special handling of some expressions.

Such expressions are:

- branching expressions, such as `IF`, explained in Listing 3. The implementation of other branching expressions, such as `And`, `Or`, `CountIf` etc. are modified as expected.
- `Closure` and `Apply`, where the result is $\top$ if the first argument is $\top$, and evaluated as the `CostEvaluator` otherwise.
- The `Map` family of functions, `HScan` and `VScan`, and `Tabulate`, all result in the value $\top$ with cost $\infty$, in case any argument is $\top$.
- Function calls, where the result is $\top$ with the cost of evaluating the arguments plus the cost of evaluating the function, in case any of the argument is $\top$. The result is always $\top$, as a $\top$ argument may be error.

```
1  public Value EvalIf(FunCall expr, Sheet sheet, int
       col, int row)
2  {
3      CostResult condition = expr.es[0].Eval(this,
           sheet, col, row);
4
5      // ...
6
7      if (condition.Value is NumberValue n0) {
8          int index = n0.value != 0.0 ? 1 : 2;
9          CostResult result = expr.expressions[index].
               Eval(this, sheet, col, row);
10         return MakeCostResult(result.Value, condition
               .Cost + result.Cost);
11     } else if (condition.Value is Top) {
12         CostResult tt = expr.es[1].Eval(this, sheet,
               col, row);
13         CostResult ff = expr.es[2].Eval(this, sheet,
               col, row);
14         var cost = (tt.Cost > ff.Cost ? tt.Cost : ff.
               Cost) + condition.Cost;
15         return MakeCostResult(Join(tt.Value,ff.Value)
               , cost);
16     } else {
17         return MakeCostResult(ErrorValue.argTypeError
               , condition.Cost);
18     }
19 }
```

**Listing 3:** Simplified C# code for abstract cost-evaluation of `IF`

To handle $\top$, the else-branch in line 11 of Listing 2 is modified as shown in Listing 3.

In this modification, if the condition is $\top$, the resulting value is a join of the values of both branches, with cost of the expensive branch with an added cost of the condition-evaluation cost plus the unit cost of the if-expression. Otherwise, the result is the result of evaluation by the `CostEvaluator` implementation.

**Table 1**
The total concrete cost, total abstract cost, overapproximation in the abstract cost evaluation, number of formula cells and the time taken to evaluate the cost of all cells in the LibreOffice Calc and EUSES spreadsheets. The cost evaluation was run twenty times and the average of those runs are shown in the **Runtime** column.

| Spreadsheet | Concrete cost | Abstract cost | Overapprox. | Formulas | Runtime |
|---|---|---|---|---|---|
| LibreOffice Calc | | | | | (runtime in *seconds*) |
| building-design | 978 520 000 | 978 520 000 | 100% | 108 332 | 33.64 |
| energy-markets | 2 175 001 469 | 2 175 001 469 | 100% | 534 507 | 3011.96 |
| grossprofit | 4 423 203 701 | 4 423 203 701 | 100% | 135 073 | 2324.62 |
| ground-water | 1 099 998 389 | 1 099 998 389 | 100% | 126 404 | 79.39 |
| stock-history | 1 230 276 358 | 1 230 276 358 | 100% | 226 503 | 85.30 |
| stocks-price | 1 165 235 199 | 1 165 235 199 | 100% | 812 693 | 1344.60 |
| EUSES | | | | | (runtime in *milliseconds*) |
| 2004_PUBLIC_BUGS_INVENTORY | 140 925 | 140 925 | 100% | 4 495 | 28.83 |
| Aggregate20Governanc#A8A51 | 723 436 | ∞ | NA | 3 546 | 154.93 |
| high_2003_belg | 11 616 516 | ∞ | NA | 12 861 | 58.56 |
| DNA | 127 029 | 127 029 | 100% | 4 715 | 15.76 |
| EUSE | 3463 | 3463 | 100% | 413 | 1.27 |
| PLANCK | 25 200 | 25 200 | 100% | 806 | 13.33 |
| 02rise | 91 581 | 91 581 | 100% | 10 316 | 26.64 |
| financial-model-spreadsheet | 20 128 | ∞ | NA | 3 115 | 10.99 |
| Financial-Projections | 31 400 | 31 994 | 101.9% | 3 649 | 11.04 |
| 2000_places_School | 9286 | 9286 | 100% | 1 375 | 2.39 |
| 2002Qvols | 10 222 | 10 222 | 100% | 2 184 | 2.35 |
| EducAge25 | 34 058 | 34 058 | 100% | 1 470 | 6.19 |
| notes5CMISB200SP04H2KEY | 156 093 | ∞ | NA | 1 557 | 103.60 |
| Test20Station20Powe#A90F3 | 15 720 | 15 720 | 100% | 2 164 | 5.59 |
| v1tmp | 6157 | 6257 | 101.62% | 1 129 | 2.06 |
| MRP_Excel | 415 529 | ∞ | NA | 4 809 | 92.16 |
| ny_emit99 | 76 010 | 76 010 | 100% | 4 352 | 24.28 |
| Time | 33 832 | 33 832 | 100% | 4 198 | 6.65 |
| WasteCalendarCalculat#A843B | 10 309 | 11 901 | 115.44% | 843 | 1.81 |
| funding | 280 702 | ∞ | NA | 1 636 | 215.05 |
| iste-cs-2003-modeling-sim | 14 919 | 14 919 | 100% | 1 991 | 6.71 |
| modeling-3 | 1292 | 1292 | 100% | 213 | 0.54 |

Taking the previous example from Section 6.2 with ⊤ instead of a numeric value as condition:

```
=IF(UNKNOWN(), SIN(1+2), COS(3))
=IF(UNKNOWN(), SIN(3), COS(1+2))
```

the result of both the above expressions is ⊤ with the cost of 6.

## 7. Results

In this section, we present our results for the concrete and abstract cost evaluators.

### 7.1. Concrete cost evaluator results

Since the concrete cost evaluator costs are proportional to the number of operations of an expression or alternatively the number of rule applications, we are not particularly interested in the precision of the costs since they are not an estimation of the actual running time of the spreadsheet. As we mentioned in Section 2.1, costs based on measurements of real machine execution could lead to more realistic costs. Instead, we are interested in how long it takes to compute the cost of each cell in a spreadsheet.

Table 1 contains the costs, number of formula cells and time taken to compute the cost of all cells in six spreadsheets from LibreOffice Calc [31] and a subset of the EUSES corpus [32]. The costs correspond to applying the $\gamma$ function to each cell address $ca$ in the spreadsheet as presented in Section 3.4.

### 7.2. Abstract cost evaluator results

In ordinary spreadsheets, there are no unknown values resulting in an abstract calculation, i.e. ⊤ values produced by the `UNKNOWN` function, so in this case the abstract cost evaluator would compute the same values and costs as the concrete cost evaluator. Therefore, in our evaluation of the abstract cost evaluator, we replace all constants in the spreadsheet with ⊤, before the abstract cost evaluator is run. The results are found in Table 1. Because some conditional values related to recursive calls are ⊤, some cost-results are ∞, because of infinite recursion. The largest overapproximation is found in `WasteCalendarCalculat#A843B`. This is caused by a large number of `IF`s, of the form: `IF(T11>-1, 0, IF(T11<1, (S11-F11)/F11, 0))` where `T11` is ⊤ in the abstract version. Other spreadsheets have either cost-balanced branchings or the most expensive branch is also taken in concrete evaluation.

### 7.3. Discussion

At a glance, we notice that there seems to be no correlation between the number of formula cells and the time taken to evaluate the cost of each cell in the spreadsheet. This is to be expected as the formula count does not tell us anything about the complexity of each one. For example, the `ny_emit99` and `Time` spreadsheets have almost the same number of formula cells but vastly different concrete costs and runtime.

## 8. Conclusion and future work

A precise cost semantics was presented in Section 3 and in Section 4. The cost evaluation semantics for Funcalc was extended to compute with unknown values in Section 5, which serves as a first step towards an approximate cost analysis, based on abstract interpretation. Finally, implementations for the concrete and abstract cost semantics were presented in Section 6.

The purpose of the cost semantics and calculations is to serve as a guide for load-balancing parallel computations in spreadsheets, e.g. via task partitioning for execution on multi-core CPUs [9] or off-loading work to GPGPUs [2]. Moreover, the evaluation and cost semantics may serve to improve the understanding of spreadsheet computations in general and the safety of and reliance on a given implementation. Also, they may be used to prove that optimizations preserve the meaning of spreadsheet computation and that these optimizations reduce the

amount of work needed to perform a computation. The cost calculations are based on counting computational steps. It would be possible to extend the cost calculations to approximate runtime execution time by measuring the execution time of each basic computational step on a given computer and use this information as a factor in the equations. Clearly such calculations are platform dependent.

The approximate abstract cost analysis is a first step towards a more general framework of abstract interpretation of spreadsheet expressions. The safety property, presented in Section 5, was established directly between the concrete cost semantics and the approximate abstract cost semantics following ideas from [27], albeit only for terminating derivations. Although the approximate abstract cost analysis can be seen as a rudimentary abstract interpretation, we did not apply the "standard" approach to establishing correctness for abstract interpretations. Usually a collecting semantics is the starting point for a sequence of more and more abstract semantics in abstract interpretation which eventually leads to an implementable analysis, i.e. a semantics-based description that can be turned into an algorithm implementing the desired analysis. In the standard setting, safety and correctness of the analysis is established via Galois connections and widening operators [33]. We expect to follow this approach when generalizing our work to a more generic framework for abstract interpretation of spreadsheets.

Due to the higher-order nature of Funcalc, another future development would be a closure analysis to improve cost estimates of function application.

Another future development may be to give a precise semantics for *depth* (also called *span* or *critical path length*) i.e. the length of the longest sequential dependence, for parallel evaluation in the sense of Blelloch [8]. This could be used as basis for an abstract interpretation to estimate *depth* in addition to the *work* defined in this paper.

Finally, one could also imagine tools, based in the formal semantics, for analyzing or verifying various aspects of spreadsheets. One such tool could be a tool to formally verify the correctness of the spreadsheet program. Another tool could guide users through performance bottlenecks in a spreadsheet and even suggest possible improvements.

## 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:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Supervision, Writing – review & editing. **Lone Leth Thomsen:** Conceptualization, Formal analysis, Investigation, Methodology, Supervision, Writing – review & editing.

## Declaration of competing interest

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.

## References


[1] Chris Scaffidi, Counts and earnings of end-user developers, https://www.linkedin.com/pulse/counts-earnings-end-user-developers-chris-scaffidi?published=t.

[2] Jim Trudeau, Collaboration and open source at AMD: LibreOffice, 2015, https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/.

[3] Mani Chandy, Concurrent programming for the masses (invited address), in: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, in: PODC '85, ACM, New York, NY, USA, ISBN: 0-89791-168-7, 1985, pp. 1–12, http://dx.doi.org/10.1145/323596.323597, URL http://doi.acm.org/10.1145/323596.323597.

[4] Andrew P. Wack, Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modeled Message Passing Environment, (Ph.D. thesis), University of Delaware, Newark, DE, USA, 1996, URL http://portal.acm.org/citation.cfm?id=269551.

[5] David Abramson, Paul Roe, Lew Kotler, Dinelli Mather, ActiveSheets: Supercomputing with spreadsheets, in: 2001 High Performance Computing Symposium (HPC '01), Advanced Simulation Technologies Conference, 2001, pp. 22–26.

[6] Amir Hirsch, Compiling and optimizing spreadsheets for FPGA and multicore execution, Massachusetts Institute of Technology, 2007, URL https://dspace.mit.edu/handle/1721.1/45983.

[7] Peter Sestoft, Spreadsheet Implementation Technology, The MIT Press, ISBN: 9780262526647, 2014.

[8] Guy E. Blelloch, Programming parallel algorithms, Commun. ACM (ISSN: 0001-0782) 39 (3) (1996) 85–97, http://dx.doi.org/10.1145/227234.227246, URL http://doi.acm.org/10.1145/227234.227246.

[9] Thomas Bøgholm, Kim G. Larsen, Marco Muniz, Bent Thomsen, Lone Leth Thomsen, Analyzing spreadsheets for parallel execution via model checking, in: Essays on the Occasion of Bernhard Steffen's 60th Birthday (Lecture Notes in Computer Science, Vol. 11200), Springer-Verlag, 2018, http://dx.doi.org/10.1007/978-3-030-22348-9_3.

[10] Alexander Asp Bock, Thomas Bøgholm, Peter Sestoft, Bent Thomsen, Lone Leth Thomsen, On the semantics for spreadsheets with sheet-defined functions, J. Comput. Lang. (ISSN: 2590-1184) 57 (2020) 100960, http://dx.doi.org/10.1016/j.cola.2020.100960, URL http://www.sciencedirect.com/science/article/pii/S2590118420300204.

[11] Simon Peyton-Jones, Alan Blackwell, Margaret Burnett, A user-centred approach to functions in excel, in: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, in: ICFP '03, ACM, ISBN: 1-58113-756-7, 2003, pp. 165–176, http://dx.doi.org/10.1145/944705.944721, URL http://doi.acm.org/10.1145/944705.944721.

[12] Gustavo Gómez, Yanhong A. Liu, Automatic time-bound analysis for a higher-order language, in: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, in: PEPM '02, ACM, New York, NY, USA, ISBN: 1-58113-455-X, 2002, pp. 75–86, http://dx.doi.org/10.1145/503032.503039, URL http://doi.acm.org/10.1145/503032.503039.

[13] Mads Rosendahl, Automatic complexity analysis, in: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, in: FPCA '89, ACM, New York, NY, USA, ISBN: 0-89791-328-0, 1989, pp. 144–156, http://dx.doi.org/10.1145/99370.99381, URL http://doi.acm.org/10.1145/99370.99381.

[14] Ben Wegbreit, Mechanical program analysis, Commun. ACM (ISSN: 0001-0782) 18 (9) (1975) 528–539, http://dx.doi.org/10.1145/361002.361016, URL http://doi.acm.org/10.1145/361002.361016.

[15] D. Le Metayer, Mechanical analysis of program complexity, in: ACM SIGPLAN Notices, Vol. 20, (7) ACM, 1985, pp. 69–73.

[16] David Sands, Complexity analysis for a lazy higher-order language, in: Functional Programming, Springer, 1990, pp. 56–79, http://dx.doi.org/10.1007/978-1-4471-3166-3_5.

[17] David Sands, A naive time analysis and its theory of cost equivalence, J. Logic Comput. 5 (4) (1995) 495–541, http://dx.doi.org/10.1093/logcom/5.4.495.

[18] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, Martin Hofmann, Static determination of quantitative resource usage for higher-order programs, in: ACM Sigplan Notices, Vol.45, (1) ACM, 2010, pp. 223–236, http://dx.doi.org/10.1145/1706299.1706327.

[19] Jan Hoffmann, Zhong Shao, Automatic static cost analysis for parallel programs, in: European Symposium on Programming Languages and Systems, Springer, 2015, pp. 132–157, http://dx.doi.org/10.1007/978-3-662-46669-8_6.

[20] Norman Danner, Daniel R. Licata, Ramyaa Ramyaa, Denotational cost semantics for functional languages with inductive types, in: ACM SIGPLAN Notices, Vol.50, (9) ACM, 2015, pp. 140–151, http://dx.doi.org/10.1145/2784731.2784749.

[21] Jay McCarthy, Burke Fetscher, Max New, Daniel Feltey, Robert Bruce Findler, A Coq library for internal verification of running-times, in: International Symposium on Functional and Logic Programming, Springer, ISBN: 978-3-319-29604-3, 2016, pp. 144–162, http://dx.doi.org/10.1007/978-3-319-29604-3_10.

[22] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, Jan Hoffmann, Relational cost analysis, Vol. 52, (1) ACM, 2017, http://dx.doi.org/10.1145/3093333.3009858.

[23] Peng Wang, Di Wang, Adam Chlipala, TiML: a functional language for practical complexity analysis with invariants, Proc. ACM Prog. Lang. 1 (OOPSLA) (2017) 79, http://dx.doi.org/10.1145/3133903.

[24] Patrick M. Sansom, Simon L. Peyton Jones, Time and space profiling for non-strict, higher-order functional languages, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995, pp. 355–366.

[25] Guy Blelloch, John Greiner, Parallelism in sequential functional languages, in: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, 1995, pp. 226–237.

[26] Ankush Das, Jan Hoffmann, ML for ML: learning cost semantics by experiment, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2017, pp. 190–207, http://dx.doi.org/10.1007/978-3-662-54577-5_11.

[27] David A. Schmidt, Trace-based abstract interpretation of operational semantics, LISP Symb. Comput. (ISSN: 1573-0557) 10 (3) (1998) 237–271, http://dx.doi.org/10.1023/A:1007734417713.

[28] Martin Bodin, Thomas Jensen, Alan Schmitt, Certified abstract interpretation with pretty-big-step semantics, in: Proceedings of the 2015 Conference on Certified Programs and Proofs, 2015, pp. 29–40.

[29] David Darais, Nicholas Labich, Phuc C. Nguyen, David Van Horn, Abstracting definitional interpreters, 2017, http://dx.doi.org/10.1145/3110256, CoRR abs/1707.04755 arXiv:1707.04755.

[30] Daniel P. Friedman, Anurag Mendhekar, Tutorial: Using an Abstracted Interpeter to Understand Abstract Interpretation, Indiana University, 2003, Course notes for CSCI B621.

[31] The Document Foundation, LibreOffice Calc, https://www.libreoffice.org/discover/calc/.

[32] Marc Fisher, Gregg Rothermel, The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms, SIGSOFT Softw. Eng. Notes (ISSN: 0163-5948) 30 (4) (2005) 1–5, http://dx.doi.org/10.1145/1082983.1083242, URL http://doi.acm.org/10.1145/1082983.1083242.

[33] Flemming Nielson, Hanne Riis Nielson, Infinitary control flow analysis: a collecting semantics for closure analysis, in: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1997, pp. 332–345.