Operational Semantics of λ_ω

ANONYMOUS AUTHOR(S)

CCS Concepts: • **Theory of computation** → **Operational semantics**;

Additional Key Words and Phrases: ordinals, arrays, semantics, functional languages

ACM Reference format:
Anonymous Author(s). 2017. Operational Semantics of λ_ω. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 13 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this document we present syntax and formal semantics for the λ_ω — a functional language with native support for arrays indexed by ordinal numbers. Note that in the paper that uses this semantics, several refinements of the language are introduced: λ_α, λ_α^∞, λ_ω without filters and finally λ_ω with filters which is described in this document. Semantics for the intermediate refinements can be obtained from the semantics of λ_ω by removing the rules for the syntactic constructs that are not within the syntax of the given refinement.

1 SYNTAX DEFINITIONS AND INFORMAL SEMANTICS

We define the syntax of λ_ω in Fig. 1. λ_ω is an applied λ-calculus with built-in support for ordinals. Its core are the standard constructs, *i.e.* constants, variables, abstractions and applications. Constants include booleans, ordinals and arrays that can be nested for expressing multi-dimensional immutable arrays. Conditionals are supported the usual way and recursive definitions can be made using letrec-constructs. Primitive operations include the usual set of scalar operations, including arithmetic operations such as +, −, *, /, *etc.*, and comparisons <, <=, =, *etc.* Arithmetic and comparison works on ordinal numbers.

Arrays can be constructed by using potentially nested sequences of scalars in square brackets. For example, $[1, 2, 3, 4]$ denotes a four-element vector, while $[[1, 2], [3, 4]]$ denotes a two-by-two-element matrix.

The dual of array construction is a built-in operation for element selection, denoted by a dot symbol, used as an infix binary operator between an array to select from, and a valid index into that array. A valid index is a vector containing as many elements as the array has dimensions; otherwise it is undefined.

reduce combinator is a variant of *foldl*, extended to allow for multi-dimensional arrays instead of lists. *reduce* takes a binary function, a neutral element and an array of values to be reduced.

The *imap*-construct starts out with the keyword *imap*, followed by a description of the result shape, which can be either a single expression that evaluates to a vector or two expressions separated by a bar that both evaluate to vectors. The latter is used when expressions evaluated within the *imap* are non-scalar, where the right hand side of the bar describes the shape of such an expression. After the shape description a curly bracket precedes the definition of the mapping function. This function can be defined piecewise by providing a set of index-range expression pairs. We demand that the set of index ranges constitutes a partitioning of the overall index space defined through the result shape expression, *i.e.* their union covers the entire index space and the index ranges are mutually disjoint. We refer to such index ranges as *generators* (rule g in Fig. 1), and we call a pair of a generator and its subsequent expression a *partition*. Each generator defines an index set and a variable (denoted by x in rule g in Fig. 1) which serves as the formal parameter of the function to be mapped over the index set. Generators can be defined in two ways: by means of two expressions which must evaluate to vectors of the same shape,

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{array}{rcll}
e & ::= & c & \text{(constants)} \\
 & | & x & \text{(variables)} \\
 & | & \lambda x.e & \text{(abstractions)} \\
 & | & e\ e & \text{(applications)} \\
 & | & e.e & \text{(selections)} \\
 & | & \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e & \text{(conditionals)} \\
 & | & \mathit{letrec}\ x = e\ \mathit{in}\ e & \text{(recursive let binding)} \\
 & | & \mathit{imap}\ s \begin{cases} g_1: & e_1, \\ & \dots \\ g_n: & e_n \end{cases} & \text{(index map)} \\
 & | & \mathit{reduce}\ e\ e\ e & \text{(reduce)} \\
 & | & \mathit{filter}\ e\ e & \text{(filter)} \\
 & | & e + e, \dots & \text{(scalar binary operations)} \\
 & | & |e| & \text{(shape operation)} \\
 & | & \mathit{islim}\ e & \text{(limit ordinal)} \\
 & & & \\
s & ::= & e & \text{(scalar imap)} \\
 & | & e|e & \text{(generic imap)} \\
g & ::= & (e <= x < e) & \text{(index set)} \\
 & | & \_(x) & \text{(full index set)} \\
c & ::= & 0, 1, \dots, \omega, \omega + 1, \dots & \text{(numbers)} \\
 & | & \text{true, false} & \text{(booleans)} \\
 & | & [e, \dots, e] & \text{(strict immutable arrays)}
\end{array}
$$

Fig. 1. The syntax of λ_ω

constituting the lower and upper bounds of the index set, or by using the underscore notation which is syntactic sugar for the following expansion rule:

$$
\texttt{(}\textbf{imap}\ \texttt{s \{ \_(iv) ...)} \equiv \texttt{(}\textbf{imap}\ \texttt{s \{ } \underbrace{[0, \dots, 0]}_{n}\ \texttt{<= iv < s: ...)}
$$

assuming that $|s| = [n]$. The variable name of a generator can be referred to in the expression of the corresponding partition.

2 A FORMAL SEMANTICS FOR λ_ω

In λ_ω evaluated arrays are described as pairs of shape and element tuples. The shape tuple consists of numbers, and the element tuple consists of numbers, booleans or functions closures. For denoting pairs and tuples, as well as element selection and concatenation on them we use the following notations:

$$
\vec{a} = \langle a_1, \dots, a_n \rangle \implies \vec{a}_i = a_i \qquad \langle a_1, \dots, a_n \rangle +\!\!+ \langle b_1, \dots, b_m \rangle = \langle a_1, \dots, a_n, b_1, \dots, b_m \rangle
$$

To denote the product of a tuple of numbers we use the following notation:

$$
\vec{s} = \langle s_1, \dots, s_n \rangle \implies \otimes\vec{s} = s_n \cdot \dots \cdot s_1 \cdot 1
$$

Note that we multiply elements in the reverse order, which is important as ordinal multiplication is not commutative. Also, when the tuple is empty its product is one. Arrays are rectangular, and the shape vector determines the extent of every axis. The number of elements of each array is finite. Element vectors contain all the elements in a linearised form. While the reader can assume row-major order, formally, it suffices that a fixed linearisation function $F_{\vec{s}}$ exists which, given a shape vector $\vec{s} = \langle s_1, \dots, s_n \rangle$, is a bijection between indices $\{\langle 0, \dots, 0 \rangle, \dots, \langle s_1 - 1, \dots, s_n - 1 \rangle\}$ and offsets of the element vector: $\{1, \dots, \otimes\vec{s}\}$. Consider as an example the array $[[1, 2], [3, 4]]$, with F being row-major order. This array is evaluated into the shape-tuple element-tuple pair $\langle\langle 2, 2 \rangle, \langle 1, 2, 3, 4 \rangle\rangle$. Scalar constants are arrays with empty shapes. We have 5 evaluates to $\langle\langle\rangle, \langle 5 \rangle\rangle$. The same holds for booleans and function closures: true evaluates to $\langle\langle\rangle, \langle \text{true} \rangle\rangle$ and $\lambda x.e$ evaluates to $\langle\langle\rangle, \langle [\![\lambda x.e, \rho]\!] \rangle\rangle$.

F is an invariant to the presented semantics. In finite cases the usual choices of F are row-major order or column-major order. In infinite cases this might be not the best option, and one could consider space-filling curves instead. Note though that selections assume row-major order on arrays: $[[1, 2], [3, 4]].[1, 0] = 3$, but the offset into linearised vector, $F_{\langle 2,2 \rangle}(\langle 1, 0 \rangle)$, can be anything from $\{1, \dots, 4\}$. The inverse of F is denoted as $F^{-1}_{\vec{s}}$ and for every legal offset $\{1, \dots, \otimes\vec{s}\}$ it returns an index vector for that offset.

Formal definitions. To define the operational semantics of λ_ω, we use a *natural semantics* similar to the one described in [Kahn 1987]. To make sharing more visible, instead of a single environment ρ that maps names to values, we introduce a concept of storage; environments map names to pointers and storage maps pointers to values. Environments are denoted by ρ and are ordered lists of name-pointer pairs. Storage is denoted by S and consists of an ordered list of pointer-value pairs.

Formally, we construct storage and environments using the following notation:

$$S ::= \emptyset \mid S, p \mapsto v \qquad \rho ::= \emptyset \mid \rho, x \mapsto p$$

A look-up of a storage or an environment is happening *right to left* and is denoted as $S(p)$ and $\rho(x)$, respectively. Extensions are denoted with comma:

$$S' = S, p \mapsto v \qquad \rho' = \rho, x \mapsto p$$

When extending a storage, we implicitly assume that the name of the pointer is always fresh. When extending environments, we preserve the name coming from λ abstraction or *letrec*, but otherwise when we use a variable to temporarily assign a pointer, the variable name is assumed to be fresh.

Semantic judgements take the form:

$$S; \rho \vdash e \Downarrow S'; p$$

where S and ρ are initial storage and environment and e is a λ_ω expression to be evaluated. The result of this evaluation ends up in the storage S' and the pointer p points to it. To simplify notation, sometimes we will use judgements of the following form:

$$S; \rho \vdash e \Downarrow S'; p \Rightarrow v \qquad\qquad \text{to denote} \qquad\qquad S; \rho \vdash e \Downarrow S'; p \wedge S'(p) = v$$

Values. The values in this semantics are constants and closures that may contain λ term, *imap* or *filter*:

$$\langle\langle \dots \rangle, \langle \dots \rangle\rangle \qquad \langle\langle\rangle, \langle [\![\lambda x.e, \rho]\!] \rangle\rangle \qquad \left[\!\!\left[\mathit{imap}\ p_{\text{out}} | p_{\text{in}} \begin{cases} \bar{g}_1 : & e_1, \\ \dots & \\ \bar{g}_n : & e_n \end{cases} , \rho \right]\!\!\right] \qquad \left[\!\!\left[\mathit{filter}\ p_f\ p_e \begin{cases} \alpha_1 & v_r^1\ v_i^1 \\ \dots & \\ \alpha_n & v_r^n\ v_i^n \end{cases} \right]\!\!\right]$$

The abstraction closure contains the λ term and the environment where this term shall be evaluated. The *imap* closure contains pointers to frame and element shapes (p_{out} and p_{in} correspondingly), the list of partitions, where generators have been evaluated and the environment in which the *imap* shall be evaluated. The filter closure contains the pointer to the filtering function p_f, the shape of the argument we are filtering over (p_e) and the list of partitions that consist of a limit ordinal, and a pair of partial result and natural number: v_r and v_i correspondingly.

Auxiliary terms. To reduce the number of conditions in the rules we use the idea of pretty-big-step operational semantics [Charguéraud 2013] and introduce auxiliary terms:

$\boldsymbol{Gen}(x, \vec{l}, \vec{u})$ — the term that captures evaluated generator, where x is the name of the generator variable and $\vec{l}$ and $\vec{u}$ are generator lower and upper bounds correspondingly.

$\boldsymbol{reduce}_1\ p_r\ i\ p_f\ p_a$ — partial reduction, where p_r is a partial result, i is the current index and p_f and p_a are reduction function and the array we are reducing correspondingly.

$\boldsymbol{imap}_1\ p_o|p_i\ \{\vec{\imath}^1 \mapsto p_{\vec{\imath}^1}, \ldots, \vec{\imath}^n \mapsto p_{\vec{\imath}^n}\}$ — the term to represent a strict *imap* where every frame index has been evaluated. The first two arguments, p_o and p_i are pointers to frame and cell shapes. The last argument is a set which defines index-pointer bindings, for every index within the frame shape.

Meta-operators. Further in this section we use the following meta-operators:

$\mathbf{E}(v)$ Lift the internal representation of a vector or a number into a valid λ_ω expression. For example: $\mathbf{E}(5) = 5$, $\mathbf{E}(\langle 1, 2, 3\rangle) = [1, 2, 3]$, *etc.*

$\langle \vec{s}, \_ \rangle$ We use underscore to omit the part of a data structure, when binding names. For example: $S;\ p \Rightarrow \langle \vec{s}, \_ \rangle$ refers to binding the variable $\vec{s}$ to the shape of $S(p)$ which must be a constant.

2.1 Core Rules

The core rules of λ_ω are similar to most of the strict functional languages:

$$\textsc{Var}\quad \frac{x \in \rho \qquad \rho(x) \in S}{S; \rho \vdash x \Downarrow S;\ \rho(x)} \qquad\qquad \textsc{Abs}\quad \frac{}{S; \rho \vdash \lambda x.e \Downarrow S, p \mapsto \langle \langle\rangle, \langle [\![\lambda x.e, \rho]\!] \rangle\rangle\ ;\ p}$$

$$\textsc{App}\quad \frac{S; \rho \vdash e_1 \Downarrow S_1;\ p_1 \Rightarrow \langle \langle\rangle, [\![\lambda x.e, \rho_1]\!] \rangle \qquad S_1; \rho \vdash e_2 \Downarrow S_2;\ p_2 \qquad S_2; \rho_1, x \mapsto p_2 \vdash e \Downarrow S_3;\ p_3}{S; \rho \vdash e_1\ e_2 \Downarrow S_3;\ p_3}$$

$$\textsc{Prf}\quad \frac{\begin{array}{c} S; \rho \vdash e_1 \Downarrow S_1;\ p_1 \Rightarrow v_1 \\ S_1; \rho \vdash e_2 \Downarrow S_2;\ p_2 \Rightarrow v_2 \\ v_1\ \mathrm{sem}(+)\ v_2 = v \end{array}}{S; \rho \vdash e_1 + e_2 \Downarrow S_2, p \mapsto v;\ p} \qquad \textsc{If-True}\quad \frac{\begin{array}{c} S; \rho \vdash e_1 \Downarrow S_1;\ p_1 \Rightarrow \langle \langle\rangle, \langle \mathrm{true} \rangle\rangle \\ S_1; \rho \vdash e_2 \Downarrow S_2;\ p_2 \end{array}}{S; \rho \vdash \mathit{if}\, e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow S_2;\ p_2} \qquad \textsc{If-False}\quad \frac{\begin{array}{c} S; \rho \vdash e_1 \Downarrow S_1;\ p_1 \Rightarrow \langle \langle\rangle, \langle \mathrm{false} \rangle\rangle \\ S_1; \rho \vdash e_3 \Downarrow S_2;\ p_2 \end{array}}{S; \rho \vdash \mathit{if}\, e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow S_2;\ p_2}$$

Primitive functions assume availability of opaque functions sem($\oplus$), where $\oplus$ ranges over primitive operations, that give meaning to the operation. When evaluating conditionals, true and false, as any scalars in λ_ω, are represented as arrays.

The computation of constants deals with scalars and multidimensional immutable arrays.

$$\textsc{Const-Scal}\quad \frac{c\ \mathit{is\ scalar}}{S; \rho \vdash c \Downarrow S_1, p \mapsto \langle \langle\rangle, \langle c \rangle\rangle\ ;\ p} \qquad\qquad \textsc{Imm-Array-empty}\quad \frac{}{S; \rho \vdash [] \Downarrow S, p \mapsto \langle \langle 0 \rangle, \langle\rangle\rangle\ ;\ p}$$

$$\textsc{Imm-Array}\quad \frac{\begin{array}{c} n \geq 1 \qquad \overset{n}{\underset{i=1}{\forall}}\ S_i; \rho \vdash c_i \Downarrow S_{i+1};\ p_i \qquad P = \langle p_1, \ldots, p_n \rangle \qquad \mathrm{AllFiniteShape}(S_{n+1}, P) \qquad \mathrm{AllSameShape}(S_{n+1}, P) \\ S' = S_{n+1}, p_o \mapsto \langle \langle 1 \rangle, \langle n \rangle\rangle, p_i \mapsto S_{n+1}(p_1) \qquad S', \rho \vdash \mathit{imap}_1\ p_o|p_i\ \ \{\langle i{-}1 \rangle \mapsto p_i \mid i \in \{1, \ldots, n\}\} \Downarrow S'';\ p \end{array}}{S_1; \rho \vdash [c_1, \ldots, c_n] \Downarrow S'';\ p}$$

Scalar constants are directly put into the storage. For arrays we evaluate all the components and ensure that they are all of the same finite shape. Subsequently, we assemble evaluated components into an $imap_1$ for final extraction of scalar values and putting them into the data tuple.

2.1.1 IMap.

Generators of *imap*s never appear as values in the given semantics. Nevertheless, it is convenient to have a notation for evaluated generators. For that purpose we introduce $Gen(x, \vec{l}, \vec{u})$ which is a triplet that keeps a variable name, lower bound and upper bound of a generator together. The rule for the generator is defined as follows:

$$\textsc{Gen}\quad \frac{S; \rho \vdash e_1 \Downarrow S_1;\ p_1 \Rightarrow \left\langle \langle n \rangle, \vec{l} \right\rangle \qquad S_1; \rho \vdash e_2 \Downarrow S_2;\ p_2 \Rightarrow \langle \langle n \rangle, \vec{u} \rangle}{S; \rho \vdash (e_1 \leq x < e_2) \Downarrow S, p \mapsto Gen(x, \vec{l}, \vec{u});\ p}$$

We present strict and lazy versions of the *imap* semantics. The strict one forces evaluation of all the scalar elements and produces the constant array $\left\langle \vec{s}, \vec{d} \right\rangle$. The lazy version, evaluates shapes, generators and puts the overall *imap* into a closure. Evaluation of the enclosed generator expressions e_k is triggered by selections into the *imap*.

$$\textsc{IMap-Strict}\quad \frac{\begin{array}{c} S; \rho \vdash e_{\text{out}} \Downarrow S_1;\ p_{\text{out}} \Rightarrow \langle \langle d_o \rangle, \vec{s_{\text{out}}} \rangle \qquad S_1; \rho \vdash e_{\text{in}} \Downarrow S_2;\ p_{\text{in}} \Rightarrow \langle \langle d_i \rangle, \vec{s_{\text{in}}} \rangle \qquad \otimes(\vec{s_{\text{out}}} \mathbin{++} \vec{s_{\text{in}}}) < \omega \\ \hat{S}_1 = S_2 \qquad \overset{n}{\underset{i=1}{\forall}} \hat{S}_i; \rho \vdash g_i \Downarrow \hat{S}_{i+1};\ p_{g_i} \Rightarrow \bar{g}_i \qquad \text{FormsPartition}(\vec{s_{\text{out}}}, \{\bar{g}_1, \ldots, \bar{g}_n\}) \\ \bar{S}_1 = \hat{S}_{n+1} \qquad \forall (i, \vec{\imath}) \in \text{Enumerate}(\vec{s_{\text{out}}}) \exists k : \left| \begin{array}{l} \vec{\imath} \in \bar{g}_k \ \wedge\ \bar{g}_k = Gen(x_k, \_, \_) \\ \bar{S}_i, p \mapsto \langle \langle d_o \rangle, \vec{\imath} \rangle; \rho, x_k \mapsto p \vdash e_k \Downarrow \bar{S}'_i;\ p_{\vec{\imath}} \\ \bar{S}'_i; \rho, x \mapsto p_{\vec{\imath}} \vdash |x| \Downarrow \bar{S}_{i+1};\ p'_{\vec{\imath}} \Rightarrow \langle \langle d_i \rangle, \vec{s_{\text{in}}} \rangle \end{array} \right. \\ \bar{S}_{\otimes \vec{s_{\text{out}}}+1}, \rho \vdash imap_1\ p_{\text{out}} | p_{\text{in}}\ \ \{\vec{\imath} \mapsto p_{\vec{\imath}} \mid (\_, \vec{\imath}) \in \text{Enumerate}(\vec{s_{\text{out}}})\} \Downarrow S';\ p \end{array}}{S; \rho \vdash imap\ e_{\text{out}} | e_{\text{in}} \begin{cases} g_1 : & e_1, \\ \ldots & \\ g_n : & e_n \end{cases} \Downarrow S';\ p}$$

Strict evaluation of an *imap* happens in three steps. First, we compute shapes and generators, making sure that generators form a partition of $\vec{s_{\text{out}}}$ (FormsPartition is responsible for this) Secondly, for every valid index defined by the frame shape, we find a generator that includes the given index (denoted $\vec{\imath} \in \bar{g}_k$). We evaluate the generator expression e_k, binding the generator variable x_k to the corresponding index value and check that the result has the same shape as p_{in}. Finally, we combine evaluated expressions for every index of the frame shape into $imap_1$ for further extraction of scalar values.

$$\textsc{IMap-Lazy}\quad \frac{\begin{array}{c} S; \rho \vdash e_{\text{out}} \Downarrow S_1;\ p_{\text{out}} \Rightarrow \langle \langle \_ \rangle, \vec{s_{\text{out}}} \rangle \qquad S_1; \rho \vdash e_{\text{in}} \Downarrow S_2;\ p_{\text{in}} \Rightarrow \langle \langle \_ \rangle, \_ \rangle \\ \hat{S}_1 = S_2 \qquad \overset{n}{\underset{i=1}{\forall}} \hat{S}_i; \rho \vdash g_i \Downarrow \hat{S}_{i+1};\ p_{g_i} \Rightarrow \bar{g}_i \qquad \text{FormsPartition}(\vec{s_{\text{out}}}, \{\bar{g}_1, \ldots, \bar{g}_n\},) \end{array}}{S; \rho \vdash imap\ e_{\text{out}} | e_{\text{in}} \begin{cases} g_1 : & e_1, \\ \ldots & \\ g_n : & e_n \end{cases} \Downarrow \hat{S}_{n+1}, p \mapsto \left[\!\!\left[imap\ p_{\text{out}} | p_{\text{in}} \begin{cases} \bar{g}_1 : & e_1, \\ \ldots & ; \rho \\ \bar{g}_n : & e_n \end{cases} \right]\!\!\right];\ p}$$

When evaluating an *imap* lazily, we also evaluate shapes and generators, check that generators partition the frame shape p_{out} and create the closure where generators, generator expressions, the frame and cell shapes that we have evaluated are stored.

Finally we define the evaluation of the auxiliary term $imap_1$.

IMAP$_1$

$$\frac{\begin{array}{c} S(p_{\text{out}}) = \langle\langle n\rangle, \vec{s_{\text{out}}}\rangle \qquad S(p_{\text{in}}) = \langle\langle m\rangle, \vec{s_{\text{in}}}\rangle \qquad \vec{s} = \vec{s_{\text{out}}} \mathbin{++} \vec{s_{\text{in}}} \\ \forall(i,\vec{\imath}) \in \text{Enumerate}(\vec{s}\,): \left| \begin{array}{l} (\vec{\imath}_{\text{out}}, \vec{\imath}_{\text{in}}) = \text{Split}(n, \vec{\imath}\,) \\ S_i; \rho, x \mapsto p_{\vec{\imath}_{\text{out}}} \vdash x.\mathbf{E}\,(\vec{\imath}_{\text{in}}) \Downarrow S_{i+1}; p'_i \Rightarrow a_i \end{array}\right. \qquad A = \langle\vec{s}, \langle a_1, \ldots, a_{\otimes\vec{s}}\rangle\rangle \end{array}}{S; \rho \vdash imap_1\ p_{\text{out}}|p_{\text{in}}\ \left\{iv_1 \mapsto p_{iv_1}, \ldots, iv_{\text{n}} \mapsto p_{iv_{\text{n}}}\right\} \Downarrow S', p \mapsto A;\ p}$$

We compute the shape of the entire *imap* as $\langle\langle n{+}m\rangle, \vec{s_{\text{out}}} \mathbin{++} \vec{s_{\text{in}}}\rangle$. For every index tuple within this shape the first n elements will correspond to the frame index, and the rest to the cell index. Within the third argument of the $imap_1$ we find a pointer that corresponds to the frame index. We evaluate selection at the cell index into this pointer to obtain a scalar value. Finally all these scalars are combined into the array A.

2.1.2 *Filter.*

Filters operate on 1-dimensional arrays. Filters behave differently depending on the shape of the argument array. In case it is finite, filters are strict; otherwise they are lazy. In contrast to *imap* the choice between strict and lazy semantics is fixed. In the strict case we evaluate the argument array and the predicate function. Then we apply this function to every element of the array, obtaining a mask for the entire array. Using this mask, we combine the elements of the array we are filtering into a constant array. The rule describing this process follows.

FILTER-STRICT

$$\frac{\begin{array}{c} S; \rho \vdash f \Downarrow S_1;\ p_f \qquad S_1; \rho \vdash a \Downarrow S_2;\ p_a \Rightarrow A \qquad S_2, x_a \mapsto p_a; \rho \vdash |x_a| \Downarrow S_3;\ p_s \Rightarrow \langle\langle\_\rangle, \vec{s}\,\rangle \\ \otimes\vec{s} < \omega \qquad S_3; \rho, x_f \mapsto p_f, x_a \mapsto p_a \vdash \big(imap\,\mathbf{E}\,(\vec{s}\,)\,|[]\ \left\{[0] <= iv < \mathbf{E}\,(\vec{s}\,):\ x_f\ (a.iv)\right\}\big) \Downarrow S_4; p_m \Rightarrow M \\ c = \text{CountTrue}(M) \qquad \overset{c}{\underset{i=1}{\forall}} : k \equiv F_{\langle c\rangle}(\langle i\rangle) \wedge a'_k = \text{FindTrue}^i(A, M) \qquad A' = \big\langle\langle c\rangle, \big\langle a'_1, \ldots, a'_c\big\rangle\big\rangle \end{array}}{S; \rho \vdash \mathit{filter}\ f\ a \Downarrow S_4, p \mapsto A';\ p}$$

The *imap* that computes the p_m mask is strict. We use two helper functions operating on constant arrays: CountTrue(a) which counts the number of values $\langle\langle\rangle, \langle \mathit{true}\rangle\rangle$ in the array a; and $\text{FindTrue}^i(a, m)$ which finds the index of the i-th value $\langle\langle\rangle, \langle \mathit{true}\rangle\rangle$ in the mask array m and selects the array a on the given index.

Filters applied to infinite arrays result in closures. Computations within such closures are triggered during selections.

FILTER

$$\frac{\begin{array}{c} S; \rho \vdash f \Downarrow S_1;\ p_f \qquad S_1 \rho \vdash a \Downarrow S_2;\ p_a \\ S_2, x_a \mapsto p_a; \rho \vdash |x_a| \Downarrow S_3;\ p_s \Rightarrow \langle\langle\_\rangle, \vec{s}\,\rangle \qquad \otimes\vec{s} \geq \omega \qquad S' = S_3, p_{\text{res}} \mapsto \left[\!\left[\mathit{filter}\, p_f\ p_a \left\{ 0 \quad \langle\langle 0\rangle, \langle\rangle\rangle\ \ 0 \right.\right]\!\right] \end{array}}{S; \rho \vdash \mathit{filter}\ f\ a \Downarrow S';\ p_{\text{res}}}$$

The closure of a filter contains a pointer to the evaluated array, the pointer to the evaluated predicate function and the first partition for the index 0. This partition has an empty vector as partial result and 0 as maximal index. Partial results will be updated on selections. New partitions may be added on selections on indices $\alpha{+}n$ where α is a limit ordinal and $n \in \mathbb{N}$, if the partition for α does not exist.

2.1.3 Reduction. Reductions are strict, left-associative and happen in the canonical order determined by F:

Reduce

$$\frac{\begin{array}{c} S; \rho \vdash e_{\text{neut}} \Downarrow S_1;\, p_{\text{neut}} \Rightarrow v_{\text{neut}} \\ S_1; \rho \vdash a \Downarrow S_2;\, p_a \qquad S_2; \rho \vdash f \Downarrow S_3;\, p_f \qquad S_3, p_r \mapsto v_{\text{neut}}; \rho \vdash \mathit{reduce}_1\; p_r\; 1\; p_f\; p_a \Downarrow S_4;\, p \end{array}}{S; \rho \vdash \mathit{reduce}\; f\; e_{\text{neut}}\; a \Downarrow S_4;\, p}$$

Reduce$_1$-1

$$\frac{S, a \mapsto p_a; \rho \vdash |a| \Downarrow S_1;\, p \Rightarrow \langle \langle \_ \rangle, \vec{s} \rangle \qquad n > \otimes \vec{s}}{S; \rho \vdash \mathit{reduce}_1\; p_r\; n\; p_f\; p_a \Downarrow S_1;\, p_r}$$

Reduce$_1$-2

$$\frac{\begin{array}{c} S, a \mapsto p_a; \rho \vdash |a| \Downarrow S_1;\, p \Rightarrow \langle \langle \_ \rangle, \vec{s} \rangle \qquad n \leq \otimes \vec{s} \\ \vec{\imath} = F_{\vec{s}}^{-1}(n) \qquad S_1; \rho, f \mapsto p_f, r \mapsto p_r, a \mapsto p_a \vdash f\; r\; (a.\mathbf{E}\,(\vec{\imath})) \Downarrow S_2;\, p_r' \qquad S_2; \rho \vdash \mathit{reduce}_1\; p_r'\; (n{+}1)\; p_f\; p_a \Downarrow S_3;\, p \end{array}}{S; \rho \vdash \mathit{reduce}_1\; p_r\; n\; p_f\; p_a \Downarrow S_3;\, p}$$

2.1.4 Selections. Selections are the only constructs in our language that force updates of the lazy data structures. Selections can happen into strict arrays and *imap*/*filter* closures. The rule for strict selections follow.

Sel-strict

$$\frac{S; \rho \vdash i \Downarrow S_1;\, p_i \Rightarrow \langle \langle d \rangle, \vec{\imath} \rangle \qquad S_1; \rho \vdash a \Downarrow S_2;\, p_a \Rightarrow \langle \vec{s}, \vec{a} \rangle \qquad k = F_{\vec{s}}(\vec{\imath})}{S; \rho \vdash a.i \Downarrow S_3, p \mapsto \langle \langle \rangle, \langle \vec{a}_k \rangle \rangle;\, p}$$

We evaluate the array we are selecting from and the index vector we are selecting at. Then we compute the offset into the data vector by applying F to the index vector. Finally, we get the scalar value at the corresponding index. Note that when applying F we implicitly check that:

- the index is within bounds $1 \leq k \leq \otimes \vec{s}$, as $F_{\vec{s}}$ is undefined outside the index space bounded by $\vec{s}$; and
- the index vector and the shape vector are of the same length, which means that selections evaluate scalars and not array sub-regions.

Sel-lazy-imap

$$\frac{\begin{array}{c} S; \rho \vdash i \Downarrow S_1;\, p_i \Rightarrow \langle \langle \_ \rangle, \vec{v} \rangle \qquad S_1; \rho \vdash a \Downarrow S_2;\, p_a \Rightarrow \left[\!\left[\mathit{imap}\; p_{\text{out}} | p_{\text{in}} \begin{cases} \bar{g}_1 & e_1 \\ \dots & \\ \bar{g}_n & e_n \end{cases}, \rho' \right]\!\right] \\ S_2(p_{\text{out}}) = \langle \langle m \rangle, \_ \rangle \qquad (\vec{\imath}, \vec{\jmath}) = \text{Split}(m, \vec{v}\,) \qquad \exists k : \vec{\imath} \in \bar{g}_k \qquad \bar{g}_k = \mathit{Gen}(x_k, \_, \_) \\ S_2, p \mapsto \mathbf{E}\,(\vec{\imath}); \rho', x_k \mapsto p \vdash e_k \Downarrow S_3;\, p_{\vec{\imath}} \qquad S_3; \rho', x \mapsto p_{\vec{\imath}} \vdash x.\mathbf{E}\,(\vec{\jmath}) \Downarrow S_4;\, p \qquad S_5 = \text{UpdateIMap}(S_4, p_a, \vec{\imath}, p_{\vec{\imath}}) \end{array}}{S; \rho \vdash a.i \Downarrow S_5;\, p}$$

Selections into lazy *imap*s happen at indices that are of the same length as concatenation of the *imap* frame and cell shapes. This means that the index the *imap* is selected at has to be split into frame and shape sub-indices: $\vec{\imath}$ and $\vec{\jmath}$ correspondingly. Given that $\bar{g}_k$ contains $\vec{\imath}$, we evaluate e_k with x_k being bound to $\vec{\imath}$. As this value may be non-scalar, we evaluate selection into it at $\vec{\jmath}$. Finally, the evaluated generator expression is saved within the *imap* closure. This step is performed by the helper function UpdateIMap.

Let us denote the enclosed filter we are selecting from as follows:

$$A = \left[\!\!\left[\mathit{filter}\, p_f\ p_e \begin{cases} \alpha_1 & r_1\ i_1 \\ \dots & \\ \alpha_n & r_n\ i_n \end{cases} \right]\!\!\right]$$

Let us also assume that ξ is a limit ordinal and n is a natural number. In this case selection into lazy filters can be described using the following rules.

$$\textsc{Sel-Lazy-Filter-1}\quad \frac{\begin{array}{c} S;\rho \vdash a \Downarrow S_1;\ p_a \Rightarrow A \qquad S_1;\rho \vdash idx \Downarrow S_2;\ p_{idx} \Rightarrow \langle\langle 1\rangle, \langle \xi + n\rangle\rangle \qquad S_2;\rho, x \mapsto p_a \vdash |x| \Downarrow S_3;\ p_s \\ \langle\langle 1\rangle, \langle l\rangle\rangle = S_3(p_e) \qquad \xi < l \qquad \xi \in S_3(p_a) \qquad r_\xi = \langle\langle m\rangle, \langle c_1, \dots, c_m\rangle\rangle \qquad n < m \end{array}}{S;\rho \vdash a.idx \Downarrow S_3, p \mapsto \langle\langle\rangle, \langle c_{n+1}\rangle\rangle\ ;\ p}$$

$$\textsc{Sel-Lazy-Filter-2}\quad \frac{\begin{array}{c} S;\rho \vdash a \Downarrow S_1;\ p_a \Rightarrow A \qquad S_1;\rho \vdash idx \Downarrow S_2;\ p_{idx} \Rightarrow \langle\langle 1\rangle, \langle \xi + n\rangle\rangle \qquad S_2;\rho, x \mapsto p_a \vdash |x| \Downarrow S_3;\ p_s \\ \langle\langle 1\rangle, \langle l\rangle\rangle = S_3(p_e) \qquad \xi < l \qquad \xi \notin S_3(p_a) \qquad S_4 = \text{AddPartition}(S_3, p_a, \xi) \qquad S_4;\rho, x \mapsto p_a \vdash x\ idx \vdash S_5;\ p \end{array}}{S;\rho \vdash a.idx \Downarrow S_5;\ p}$$

$$\textsc{Sel-Lazy-Filter-3}\quad \frac{\begin{array}{c} S;\rho \vdash a \Downarrow S_1;\ p_a \Rightarrow A \qquad S_1;\rho \vdash idx \Downarrow S_2;\ p_{idx} \Rightarrow \langle\langle 1\rangle, \langle \xi + n\rangle\rangle \\ S_2;\rho, x \mapsto p_a \vdash |x| \Downarrow S_3;\ p_s \qquad \langle\langle 1\rangle, \langle l\rangle\rangle = S_3(p_e) \qquad \xi < l \qquad \xi \in S_3(p_a) \qquad r_\xi = \langle\langle m\rangle, \langle c_1, \dots, c_m\rangle\rangle \\ n \geq m \qquad S_3;\rho, x_e \mapsto p_e \vdash x_e.\mathbf{E}\left(\left\langle i_\xi \right\rangle\right) \Downarrow S_4;\ p_{el} \qquad S_4;\rho, x_f \mapsto p_f, x_{el} \mapsto p_{el} \vdash x_f\ x_{el} \Downarrow S_5;\ p_b \\ \langle\langle\rangle, v\rangle = S_5(p_{el}) \qquad r'_\xi = \begin{cases} \langle\langle m+1\rangle, \langle c_1, \dots, c_m, v\rangle\rangle & S_5(p_b) = \langle\langle\rangle, \langle \text{true}\rangle\rangle \\ r_\xi & \text{otherwise} \end{cases} \\ S_6 = \text{UpdatePartition}(S_5, p_a, \xi, r'_\xi, i_\xi + 1) \qquad S_6;\rho, x_a \mapsto p_a, x_{idx} \mapsto p_{idx} \vdash x_a\ x_{idx} \Downarrow S_7;\ p \end{array}}{S;\rho \vdash a.idx \Downarrow S_7;\ p}$$

Selections into filter closures fall into three cases. First, if the ξ partition exists in the filter closure, notation $\xi \in S_3(p_a)$, then we check whether the partial result of this partition contains the index n. If so, we return the scalar value stored in the partial result at the offset $n + 1$. Secondly, if the enclosed filter does not have a partition for the limit ordinal ξ, we add this partition to the filter closure using AddPartition, which updates p_a by adding the $\xi \mapsto (\langle\langle 0\rangle, \langle\rangle\rangle, \xi)$ partition to the enclosed filter. Finally, if the ξ partition exists, but its partial result r_ξ has less than n elements, we evaluate selection into the enclosed array at $\left\langle i_\xi \right\rangle$, binding the evaluated result to p_{el}. In case the predicate function applied to p_{el} evaluates to true, we compute r'_ξ by concatenating r_ξ and p_{el}. Otherwise the partial result stays the same: $r'_\xi = r_\xi$. The ξ partition of the p_a is updated with the value $(r'_\xi, i_\xi + 1)$ using the UpdatePartition helper function. After that we evaluate selection into the updated filter closure once again.

Letrec. Recursive definitions are possible using the *letrec* construct which has the following semantics:

$$\textsc{Letrec}\quad \frac{S_1 = S, p \mapsto \bot \qquad \rho_1 = \rho, x \mapsto p \qquad S_1;\rho_1 \vdash e_1 \Downarrow S_2;\ p_2 \qquad S_3 = S_2[p_2/p] \qquad S_3;\rho, x \mapsto p_2 \vdash e_2 \Downarrow S_4;\ p_r}{S;\rho \vdash \mathit{letrec}\ x = e_1\ \mathit{in}\ e_2 \Downarrow S_4;\ p_r}$$

where $S[p_2/p]$ denotes substitution of the $x \mapsto p$ bindings inside of the enclosed environments with $x \mapsto p_2$, where x is any legal variable name.

2.1.5 Shape. In order to maintain rank and shape polymorphism it suffices to define how to compute the shape of a value at runtime. Further we define how to compute shapes of constants, λ-closures, *imap*s and *filter*s.

$$\textsc{Shape-Const}\quad \frac{S; \rho \vdash x \Downarrow S_1;\ p_x \Rightarrow \langle \langle s_1, \ldots, s_n \rangle, \_ \rangle}{S; \rho \vdash |x| \Downarrow S_1, p \mapsto \langle \langle n \rangle, \langle s_1, \ldots, s_n \rangle \rangle ;\ p} \qquad \textsc{Shape-Abs}\quad \frac{S; \rho \vdash x \Downarrow S_1;\ p \Rightarrow \langle \langle \rangle, \llbracket \lambda x'.e, \rho' \rrbracket \rangle}{S; \rho \vdash |x| \Downarrow S_1, p \mapsto \langle \langle 0 \rangle, \langle \rangle \rangle ;\ p}$$

$$\textsc{Shape-IMap}\quad \frac{\begin{array}{c} S; \rho \vdash x \Downarrow S_1;\ p_x \Rightarrow \llbracket imap\ p_{\text{out}} | p_{\text{in}}\ \_, \rho' \rrbracket \\ \langle \langle n \rangle, \vec{s} \rangle = S_1(p_{\text{out}}) \qquad \langle \langle m \rangle, \vec{u} \rangle = S_1(p_{\text{in}}) \qquad A = \langle \langle m + n \rangle, \vec{s} +\!\!+ \vec{u} \rangle \end{array}}{S; \rho \vdash |x| \Downarrow S_1, p \mapsto A;\ p}$$

Shapes of constants can be straightforwardly extracted from the constant value; shapes of λ functions are empty; the shape of an *imap* is a concatenation of its frame and cell shapes.

Shapes of infinite filters depend on the length of the argument array: if the length is a limit ordinal ξ then the shape of the result is also ξ.

$$\textsc{Shape-Filter-1}\quad \frac{S; \rho \vdash x \Downarrow S_1;\ p_x \Rightarrow \llbracket \mathit{filter}\ p_f\ p_e\ \_ \rrbracket \qquad S_1; \rho, e \mapsto p_e \vdash |e| \Downarrow S_2;\ p_s \Rightarrow \langle \langle 1 \rangle, \langle \xi \rangle \rangle}{S; \rho \vdash |x| \Downarrow S_1, p \mapsto \langle \langle 1 \rangle, \langle \xi \rangle \rangle ;\ p}$$

Otherwise, if the shape of the argument array is $\xi + n$, where $n \in \mathbb{N}$, we force filtering of the elements at indices from $\langle \xi \rangle$ to $\langle \xi + n - 1 \rangle$. Finally, the shape of the filter is ξ plus the number of elements in the r_ξ.

$$\textsc{Shape-Filter-2}\quad \frac{\begin{array}{c} S; \rho \vdash x \Downarrow S_1;\ p_x \Rightarrow \llbracket \mathit{filter}\ p_f\ p_e\ \_ \rrbracket \qquad S_1; \rho, e \mapsto p_e \vdash |e| \Downarrow S_2;\ p_s \Rightarrow \langle \langle 1 \rangle, \langle \xi + n \rangle \rangle \\ \bar{S}_0 = S_2 \qquad \overset{n-1}{\underset{i=0}{\forall}} : \bar{S}_i; \rho, e \mapsto p_e, f \mapsto p_f \vdash f\ \ (e.\mathbf{E}(\langle \xi + i \rangle)) \Downarrow \bar{S}_{i+1};\ p_i \qquad c = \sum_{i=0}^{n-1} (\bar{S}_n(p_i) = \langle \langle \rangle, \text{true} \rangle) \end{array}}{S; \rho \vdash |x| \Downarrow \bar{S}_n, p \mapsto \langle \langle 1 \rangle, \langle \xi + c \rangle \rangle ;\ p}$$

2.1.6 Limit ordinal. The builtin *islim* operation ensures that the argument is a scalar and checks whether it is a limit ordinal or not:

$$\textsc{LimitOrdinal}\quad \frac{S; \rho \vdash e \Downarrow S_1;\ p_e \Rightarrow \langle \langle \rangle, \langle \xi \rangle \rangle \qquad c = \begin{cases} \langle \langle \rangle, \langle \text{true} \rangle \rangle & \xi \text{ is a limit ordinal} \\ \langle \langle \rangle, \langle \text{false} \rangle \rangle & \text{otherwise} \end{cases}}{S; \rho \vdash \mathit{islim}\ e \Downarrow S_1, p \mapsto c;\ p}$$

As Cantor Normal Form (CNF) is used to represent ordinals, the check whether a value is a limit ordinal is straightforward. First, we check that $\xi \geq \omega$. Secondly, that the coefficient at ω^0 is zero in the CNF of ξ.

3 AUXILIARY FUNCTIONS

3.1 Tuple operations

A function to split a vector into two parts at the position k.

$$\text{Split}(k, \langle a_1, \ldots, a_n \rangle) = (\langle a_1, \ldots, a_k \rangle, \langle a_{k+1}, \ldots, a_n \rangle)$$

The $\triangleleft_d$ operator takes $d-1$ elements from the first tuple and all the elements starting from the index d from the second tuple. Both tuples are of the same length.

$$\langle a_1, \ldots, a_n \rangle \triangleleft_d \langle b_1, \ldots, b_n \rangle = \langle a_1, \ldots, a_{d-1}, b_d, \ldots, b_n \rangle$$

The $\diamond_d$ operator updates the d-th element of the first tuple with the d-th element of the second tuple.

$$\langle a_1, \ldots, a_d, \ldots, a_n \rangle \diamond_d \vec{b} = \left\langle a_1, \ldots, \vec{b}_d, \ldots, a_n \right\rangle$$

3.2 Partition operations

Partitions are pairs of tuples of the same length, where each tuple consists of numbers. To simplify the usage of partition-related auxiliary functions we will let those functions to operate on the evaluated generators, but we will ignore the variable.

First, we can find out whether two partitions intersect.

$$Gen(\_, \vec{l}, \vec{u}) \cap Gen(\_, \vec{l}', \vec{u}') = \begin{cases} Gen(\_, \vec{l}_\uparrow, \vec{u}_\downarrow) & \begin{array}{l} \vec{l}_\uparrow < \vec{u}_\downarrow \\ \wedge Gen(\_, \vec{l}_\uparrow, \vec{u}_\downarrow) \in Gen(\_, \vec{l}, \vec{u}) \\ \wedge Gen(\_, \vec{l}_\uparrow, \vec{u}_\downarrow) \in Gen(\_, \vec{l}', \vec{u}') \end{array} \\ \emptyset & \text{otherwise} \end{cases}$$

where $\vec{l}_\uparrow$ is component-wise maximum of $\vec{l}$ and $\vec{l}'$ and $\vec{u}_\downarrow$ is component-wise minimum of $\vec{u}$ and $\vec{u}'$. Two partitions intersect if $(\vec{l}_\uparrow, \vec{u}_\downarrow)$ is not empty and belongs to both partitions.

If a partition p_1 lies within p_2, we can define a split of p_2 over p_1 into $2d+1$ sub-partitions, where d is the dimensionality of partitions. The overall idea can be understood from 1-d case. Assume that the lower and upper bounds of p_2 are $(\langle 0 \rangle, \langle 10 \rangle)$ and $(\langle 2 \rangle, \langle 4 \rangle)$ for p_2. As it can be seen p_1 can be split in three parts: $(\langle 0 \rangle, \langle 2 \rangle)$, $p_1 = (\langle 2 \rangle, \langle 4 \rangle)$ and $(\langle 4 \rangle, \langle 10 \rangle)$. In the multi-dimensional case, the same idea can be applied for every dimension.

Let us define the split of the partition p' over p across the dimension d. Assume that $p = Gen(\_, \vec{l}, \vec{u})$ and $p' = Gen(\_, \vec{l}', \vec{u}')$:

$$\text{splitd}_l(\vec{l}', \vec{u}', \vec{l}, \vec{u}, d) = Gen(\_, \quad \vec{l} \triangleleft_d \vec{l}', \quad (\vec{u} \triangleleft_{d+1} \vec{u}') \diamond_d \vec{l})$$

and

$$\text{splitd}_r(\vec{l}', \vec{u}', \vec{l}, \vec{u}, d) = Gen(\_, \quad (\vec{l} \triangleleft_{d+1} \vec{l}') \diamond_d \vec{u}, \quad \vec{u} \triangleleft_d \vec{u}')$$

If we do such a split for every dimension, and we include partition intersection we will get a partition split:

$$\text{PartSplit}(p', p) = \bigcup_{i=1}^{n} \left(\left\{ \text{splitd}_l(\vec{l}', \vec{u}', \vec{l}, \vec{u}, d), \text{splitd}_r(\vec{l}', \vec{u}', \vec{l}, \vec{u}, d) \right\} \right) \cup \{p \cap p'\}$$

Finally, the function we use when evaluating *imap*s that checks whether the list of generators partition the index-space defined by the shape of the *imap*.

$$\text{FormsPartition}\,(\vec{s}, \{p_1, \ldots, p_n\}) = \forall i \neq j : p_i \cap p_j = \emptyset \wedge \bigcup_{i=1}^{n} p_i = Gen(\_, \vec{0}, \vec{s})$$

The fact that the union of partitions forms the index space defined by the shape vector $\vec{s}$ can be checked using the following inductive definition:

$$\forall i \in \{1, \ldots, n\} : R_{i+1} = \bigcup_{r_j \in R_i} \text{PartSplit}(r_j, p_i) \setminus \left(r_j \cap p_i \right)$$

as follows:

$$\bigcup_{i=1}^{n} p_i = Gen(\_, \vec{0}, \vec{s}) \iff R_1 = \left\{ Gen(\_, \vec{0}, \vec{s}) \right\} \wedge R_{n+1} = \emptyset$$

3.3 Enumerate

The Enumerate function generates an enumeration of the index space defined by $\vec{s}$ using F and returns a set of pairs, where the first element of each pair is an offset and the second element is an index vector.

$$\text{Enumerate}(\vec{s}) = \left\{ \left(i, F_{\vec{s}}^{-1}(i) \right) \mid i \in i \left\{ 1, \ldots, \otimes \vec{s} \right\} \right\}$$

3.4 UpdateIMap

The UpdateIMap function splits the $\bar{g}_k$ partition into a single-element partition containing $\vec{\imath}$ and the rest. Then the function removes $\bar{g}_k$ from the *imap* closure and adds new partitions obtained by the split. Expressions bound to every non-$\vec{\imath}$ partition remain the same as at $\bar{g}_k$, *i.e.* e_k. The expression for the $\vec{\imath}$ partition is a fresh variable x, while the enclosed environment extended with the $x \mapsto p_{\vec{\imath}}$ binding.

$$\frac{\begin{array}{c} A = \left\llbracket \mathit{imap}\; p_{\text{out}} | p_{\text{in}} \begin{cases} \ldots & \\ \bar{g}_k & e_k \\ \ldots & \end{cases}, \rho' \right\rrbracket \\ S = S_1, p_a \mapsto A, S_2 \qquad \vec{\imath} \in \bar{g}_k \qquad P = Gen(x_k,\ \vec{\imath},\ \vec{\imath}{+}\vec{1}) \qquad R = \text{PartSplit}(\bar{g}_k, P) \setminus \bar{g}_k \cap P \\ R' = \{ Gen(x_k, \vec{l}, \vec{u}) \mid Gen(\_, \vec{l}, \vec{u}) \in R \} \qquad A' = \left\llbracket \mathit{imap}\; p_{\text{out}} | p_{\text{in}} \begin{cases} \ldots & \\ P & x \\ R'_1 & e_k \\ \ldots & \\ R'_{|R'|} & e_k \\ \ldots & \end{cases}, \rho', x \mapsto p_{\vec{\imath}} \right\rrbracket \end{array}}{\text{UpdateIMap}(S, p_a, \vec{\imath}, p_{\vec{\imath}}) = S_1, p_a \mapsto A', S_2}$$

3.5 Filter-related operations

A function that counts the number of elements with value $\langle \langle \rangle, \langle \mathit{true} \rangle \rangle$ in the array.

$$\text{CountTrue}(\langle \_, \langle a_1, \ldots, a_n \rangle \rangle) = \sum_{i=1}^{n} \begin{cases} 1 & a_i = \langle \langle \rangle, \langle \mathit{true} \rangle \rangle \\ 0 & \text{otherwise} \end{cases}$$

A function that finds the index of the n-th value $\langle \langle \rangle, \langle \mathit{true} \rangle \rangle$ in the 1-dimensional array M and selects 1-dimensional array A at this index.

$$\text{FindTrue}^n(A, M) = \text{FindTrueHelper}(1, 0, n, A, M)$$

where

$$\text{FindTrueHelper}(i, m, n, \langle \_, \vec{a} \rangle, \langle \_, \vec{m} \rangle) = \begin{cases} \vec{a}_i & m = n - 1 \wedge \vec{m}_i = \langle \langle \rangle, \langle \mathit{true} \rangle \rangle \\ \text{FindTrueHelper}(i + 1, m, n, A, M) & m \neq n - 1 \wedge \vec{m}_i = \langle \langle \rangle, \langle \mathit{false} \rangle \rangle \\ \text{FindTrueHelper}(i + 1, m + 1, n, A, M) & m \neq n - 1 \wedge \vec{m}_i = \langle \langle \rangle, \langle \mathit{true} \rangle \rangle \end{cases}$$

The AddPartition function updates a filter closure by adding the $\xi \mapsto (\langle\langle 0\rangle , \langle\rangle\rangle , \xi)$ partition.

$$\frac{A = \left[\!\left[\mathit{filter}\, p_f\ p_e \begin{cases} \alpha_1 & r_1\ i_1 \\ \ldots & \\ \alpha_n & r_n\ i_n \end{cases} \right]\!\right] \qquad S = S_1, p_a \mapsto A, S_2 \qquad A' = \left[\!\left[\mathit{filter}\, p_f\ p_e \begin{cases} \alpha_1 & r_1\ i_1 \\ \ldots & \\ \alpha_n & r_n\ i_n \\ \xi & \langle\langle 0\rangle , \langle\rangle\rangle\ \xi \end{cases} \right]\!\right]}{\text{AddPartition}(S, p_a, \xi) = S_1, p_a \mapsto A', S_2}$$

The UpdatePartition function replaces partial result and index of a given partition in the filter closure with new values.

$$\frac{A = \left[\!\left[\mathit{filter}\, p_f\ p_e \begin{cases} \ldots & \\ \xi & r_\xi\ i_\xi \\ \ldots & \end{cases} \right]\!\right] \qquad S = S_1, p_a \mapsto A, S_2 \qquad A' = \left[\!\left[\mathit{filter}\, p_f\ p_e \begin{cases} \ldots & \\ \xi & r'_\xi\ i'_\xi \\ \ldots & \end{cases} \right]\!\right]}{\text{UpdatePartition}(S, p_a, \xi, r'_\xi, i'_\xi) = S_1, p_a \mapsto A', S_2}$$

4 OBSERVATIONS

Here are a few important observations regarding the semantics of λ_ω. First of all, if a program terminates, then its result has a strict shape:

Theorem 4.1.

$$\forall P \in \lambda_\omega : \emptyset; \emptyset \vdash P \Downarrow S;\ p \implies S; x \mapsto p \vdash |x| \Downarrow S';\ p' \wedge S'(p') = \langle\langle n\rangle , \langle v_1, \ldots, v_n\rangle\rangle , n \in \mathbb{N}$$

This is because constants and functions have shape by definition and *imap* evaluates shapes strictly. An attempt to define shapeless objects like:

```
letrec x = fiter (λy.true) x
```

will fail as such a program diverges.

Secondly, any number that ever appears as a result in λ_ω is less than ω^ω.

Theorem 4.2.

$$\forall P \in \lambda_\omega : \emptyset; \emptyset \vdash P \Downarrow S;\ p \implies \omega^\omega > \begin{cases} v & S(p) = \langle\langle\rangle , \langle d\rangle\rangle \\ \prod \vec{v} & S; x \mapsto p \vdash |x| \Downarrow S';\ p' \wedge S'(p') = \langle \_, \vec{v}\rangle \end{cases}$$

This is because shapes and function applications are evaluated strictly. Therefore, an attempt to construct the shape larger than ω^ω via the following series of *imap*s:

```
v ≡ imap [ω] {_(iv): ω
vv ≡ imap v {_(iv): ω
```

fails, as strict computation of v that is required in the shapes of *imap*s diverges. An attempt to create the power function directly:

```
letrec f = λr.λi.if i < ω then f r*ω i+1 else r in f 1 0
```

or via reduce:

```
reduce * 1 v
```

diverges as well because of strict evaluation. This means that in its current state all the ordinals are representable using simplified Cantor Normal form, where ω powers a_i are natural numbers:

$$\alpha = \omega^{a_1} \cdot x_1 + \cdots + \omega^{a_n} \cdot x_n + p$$

Such a restriction means that infinite arrays always have finite number of dimensions. In principle, this can be relaxed by introducing a builtin ordinal power operation, in which case we will be able to define numbers up to ϵ_0. However, at this point it is not clear whether there is a good use of such an extension. Also, this would make shapes non-strict rendering comparison of two shapes undecidable in general case.

Recursive definitions as we intuitively write them will not work as expected for transfinite numbers. Consider the following attempt to define identity function for numbers that recursively descends to 0, adding 1 at every step:

```
letrec f = λx.if x = 0 then 0 else 1 + (f (x−1)) in f α
```

For the cases when $\alpha \geq \omega$ the function will diverge, as $x - 1$ will evaluate to x. Replacing left subtraction used by default with the right one wouldn't help, as $x -_R 1$ does not exist for the cases when x is a limit ordinal. An attempt to reformulate this function into recursively ascending:

```
letrec f = λx.λn.if n = x then n else 1 + (f x (n+1)) in f α 0
```

does not help either. When $\alpha \geq \omega$ the function diverges, as there is no way to reach ω by successive application of the add 1 function. The reason for this is that normal recursion does not cover transfinite cases. To express the above computations we need to use transfinite recursion, which apart from the usual base case also defines cases for every limit ordinal. Therefore, the availability of the predicate that tells whether the number is a limit ordinal, such as *islim* in case of λ_ω, is absolutely crucial.

REFERENCES

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer Berlin Heidelberg, 41–60.

G. Kahn. 1987. Natural semantics. In *STACS 87*, FranzJ. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Lecture Notes in Computer Science, Vol. 247. Springer Berlin Heidelberg, 22–39. https://doi.org/10.1007/BFb0039592