# Flattening Nested Data Parallelism

Frederik M. Madsen

February 1, 2012


## Abstract

Based on the nested data parallel programming languages NESL, Proteus and Data Parallel Haskell existing work on vectorization is analyzed and open problems regarding work complexity are established. A general solution to the problem of distributing large vectors over many parallel instances is presented using generalized segment descriptors. A nested data parallel source language and a flat data parallel target language are formalized along with a transformation that transforms programs from the former to the latter. The transformation is proved correct with respect to value preservation and asymptotic complexity preservation by first formalizing a separate and reasonable language-based cost model for both languages. The proof is supported by empirical results.

# Contents

# 1 Introduction

## 1.1 Nested Data Parallelism

Data parallelism is a parallel programming paradigm that facilitates easy expression of large scale concurrency. It is more declarative in nature than task parallelism where the communication between parallel instances is explicit. In the data parallel paradigm parallelism is achieved by operating on an entire data set at once. The programmer specifies a computation that is applied to each element of the data set and sometimes also how to combine the results, and the compiler or interpreter than takes care of parallelizing the computation.

We will refer to the completely general form of data parallelism with no restrictions on the evaluation of each element or the combination of the results as non-uniform data parallelism because there might be no similarity in the evaluation of two different elements. Non-uniform data parallelism is well-suited for MIMD architectures.

Uniform data parallelism on the other hand requires a certain regularity on the evaluation of each element. More precisely each evaluation is required to execute the exact same instructions in the exact same order. This restriction is useful because it enables the data parallel program to run on SIMD architectures where parallelism is ususally much cheaper than in MIMD architectures. With the advent of SIMD platforms such as GPGPU's and vector instruction sets in modern personal computers, it seems prudent to invest attention in the uniform data parallel paradigm.

Nested data parallelism generalizes the idea of data parallelism to a recursive definition by explicitly stating that the parallel evaluation of each element in a data set is in itself potentially also a data parallel program, and thereby allowing expression of programs with an arbitrary deep parallel nesting structure. This paradigm allows the programmer to express highly parallelizable algorithms with an irregular recursive structure in a short and concise way that reflects the intent of the algorithm.

In a functional setting this may be expressed with the apply-to-each language construct

$$[f(x) : x \in v]$$

This construct can be read as: "Apply the function $f$ to each element of $v$ in parallel". The definition of $f$ can also contain apply-to-each constructs.

To support uniform nested data parallelism, it is necessary to transform nested data parallel programs to equivalent flat data parallel ones first. Such a flattening program transformation is called vectorization and was first proposed by Guy Blelloch and realized in the programming language NESL [Blelloch and Sabot(1990),Blelloch(1990),Blelloch(1995)]. Vectorization has later been studied and refined by others such as Jan Prins and Daniel Palmer with the language Proteus [Prins and Palmer(1993),Riely et al.(1995)Riely, Prins, and Iyer,Palmer et al.(1995)Palmer, Prins, and Westfold], and Gabriele Keller, Simon Peyton Jones and Manual Chakravarty with language Data Parallel Haskell [Keller and Simons(1996), Peyton Jones(2008), Chakravarty et al.(2007)Chakravarty, Leshchinskiy, Jones, Keller, and Marlow]. Vectorization has not changed fundamentally from the original formulation by Blelloch in the new formulations.

## 1.2 Naive Vectorization

The main idea of vectorization is to have a parallel vector-version of every primitive operation implemented efficiently in the language, such as a component-wise vector addition. Flat data parallelism is then achieved almost trivially. Nested parallelism to an arbitrary depth is achieved by repeatedly flattening any input vectors until the basic vector-version primitive can be applied to the input and then segmenting the result accordingly afterwards. This approach is made viable by the representation of nested vectors which facilitates very cheap concat and segmentation operations.

A nested vector is represented by a flat data vector with an accompanying segment descriptor that holds the nesting structure of the nested vector. As an example the nested vector

$$v = \big[[1,2,3],\ [4],\ [],\ [5,6]\big]$$

will be represented by a flat data vector

$$d = [1,2,3,4,5,6]$$

and a segment descriptor describing the length of each sub-vector

$$s_0 = [3,1,0,2]$$

This representation can be generalized to vectors of any nesting depth. For example

$$v = \Big[[]\ ,\quad \big[[1,2,3],\ [4],\ [],\ [5,6]\big]\ ,\quad \big[[7],\ [],\ [8,9,10]\big]\Big]$$

will be represented by

$$d = [1,2,3,4,5,6,7,8,9,10]$$

and

$$s_0 = [3,1,0,2,1,0,3]$$
$$s_1 = [0,4,3]$$

A general property of the segment descriptors is that the sum of the elements in a segment descriptor equals the length of the next segment descriptor (or data vector). When applying an operation in parallel the arguments are not necessarily of the same nesting depth. Consider as an example

$$[x+1 : x \in u]$$

In naive vectorization this will translate to $u +^{\mathrm{v}} [1, \ldots, 1]$ where $(+^{\mathrm{v}})$ is the vector-version of $(+)$. In other words the 1 must be replicated over the length of $u$.

Since apply-to-each may be nested it is possible to write something like

$$[[x + y : x \in xs] : y \in ys]$$

which computes addition for all pairs of $x$ and $y$ in parallel. This will translate to

$$[xs +^{\mathrm{v}} [y, \ldots, y] : y \in ys]$$

where $y$ is replicated over the length of $xs$. This will in turn translates to

$$[\overbrace{xs, \ldots, xs}^{\textit{length of ys}}](+^{\mathrm{v}})^{\mathrm{v}}[\overbrace{ys, \ldots, ys}^{\textit{length of xs}}]^{\mathrm{v}}$$

where $[\cdot]^{\mathrm{v}}$ is the vector-version of $[\cdot]$ which is similar to a transpose operation if you think of the two arguments as matrices. In order to remove $(+^{\mathrm{v}})^{\mathrm{v}}$ the arguments are flattened

$$\textit{flatten}([\overbrace{xs, \ldots, xs}^{\textit{length of ys}}]) +^{\mathrm{v}} \textit{flatten}([\overbrace{ys, \ldots, ys}^{\textit{length of xs}}]^{\mathrm{v}})$$

and the result is segmented

$$\textit{segmentate}(\textit{flatten}([\overbrace{xs, \ldots, xs}^{\textit{length of ys}}]) +^{\mathrm{v}} \textit{flatten}([\overbrace{ys, \ldots, ys}^{\textit{length of xs}}]^{\mathrm{v}}))$$

However the segmentation must know how to segmentate the result. This can be achieved by simply attaching the top-most segment descriptor of one of the results. Lets take $x = [x_1, \ldots, x_k]$ and $y = [y_1, \ldots, y_k]$. $[\overbrace{xs, \ldots, xs}^{k_2}]$ and $[\overbrace{ys, \ldots, ys}^{k_1}]^{\mathrm{v}}$ will be represented by

$$d_x = [x_1, \ldots, x_{k_1}, \ldots, x_1, \ldots, x_{k_1}] \qquad d_y = [y_1, \ldots, y_1, \ldots, y_{k_2}, \ldots, y_{k_2}]$$
$$s_0^x = [\overbrace{k_1, \ldots, k_1}^{k_2}] \qquad s_0^y = [\overbrace{k_1, \ldots, k_1}^{k_2}]$$

Notice that both segment descriptors are the same. Flattening these values simply removes the segment descriptor, so the vector-version of addition will compute

$$\begin{aligned}&[x_1, \ldots, x_{k_1}, \ldots, x_1, \ldots, x_{k_1}] +^{\mathrm{v}} [y_1, \ldots, y_1, \ldots, y_{k_2}, \ldots, y_{k_2}]\\ &\quad = [x_1 + y_1, \ldots, x_{k_1} + y_1, \ldots, x_1 + y_{k_2}, \ldots, x_{k_1} + y_{k_2}]\end{aligned}$$

Segmentation will then simply attach one of the segment descriptors which would then yield a result that is the representation of

$$[[x_1 + y_1, \ldots, x_{k_1} + y_1], \ldots, [x_1 + y_{k_2}, \ldots, x_{k_1} + y_{k_2}]]$$

It is not a coincidence that the segment descriptors are equal - it will always be the case. The reason is that anything that is not the variable bound by the apply-to-each construct (also called the iterator variable) will be replicated to match the iterator variable.

## 1.3 Ideal Cost Model

The ideal asymptotic complexity of a nested data parallel program can be expressed inductively on the constructs of the language in a natural way in terms of a work complexity and a step complexity where the work complexity characterizes the total amount of work done, and the step complexity characterizes the depth of the computation meaning the time to evaluate a program if all parallelism is realized [Blelloch(1996)]. One of the main problems in flattening nested data parallel programs is to achieve comparable cost semantics of the flattened programs. The original formulation of the representation of nested vectors and flattening nested data parallel programs poses a number of problems regarding the work complexity.

The ideal work complexity of apply-to-each should be the sum of the work done if each element is evaluated separately. So if the work complexity of $f(x_i)$ is denoted $w_i$ and $xs = [x_1, \ldots, x_k]$, then the work complexity of $[f(x) : x \in xs]$ should ideally be $\sum_{i=1}^{k} w_i$. Every other construct in the language can then be given a work complexity in a natural way to give a complete cost model. For the step complexity the summation is turned into a max, so if the step complexity of $f(x_i)$ is denoted $s_i$, then the step complexity of the apply-construct should ideally be $\max_{i=1}^{k} s_i$.

## 1.4 Problem Formulation

### 1.4.1 Replication Problem

In naive vectorization replication of values have work complexity proportional to the size of the value. This is not so much a problem for the + operation since the cost of replicating 1 is not asymptotically greater than actually computing $u +^{\mathrm{v}} [1, \ldots, 1]$. The problem arises when an argument in the original operation can have a size that surpasses the cost of performing the operation. Consider indexing $u$ in parallel over a vector of indices $is$. In the source language this could be expressed as $[u!i : i \in is]$ where ! is the indexing operator. We would expect the cost of this expression to be in the order of the length of $is$ since $u!i$ is expected to be a constant time operation. In the naive vectorization this expression will translate to $[u, \ldots, u]!^{\mathrm{v}} is$. Evaluating $[u, \ldots, u]$ is potentially extremely expensive if $u$ is big.

### 1.4.2 Provable Correct Cost Model

A more general problem is defining a realistic cost model for vectorized program and to prove that it is actually correct. Ideally the cost model should reflect the ideal cost model for the high-level nested data parallel programs with the same work and step complexities, but in practice this is not easy, and a completely general vectorization with ideal cost semantics has not been presented in any of the literature of this report. A common relaxation is to restrict programs to a form of contained programs [Blelloch(1990)] or to relax the ideal cost semantics [Riely et al.(1995)Riely, Prins, and Iyer]. The main purpose of this report is to try to find a cost model as close to the ideal cost model as possible and a vectorization such that the vectorized programs can be shown have asymptotically the same cost as the cost model dictates. We will also address the cost of each basic operation, as opposed to deriving general constraints on the work complexity of vector-versions of operations based on the work complexity of the original operation. To narrow the scope we will not consider step complexity formally but we will keep it in mind.

# 2 Analysis

The existing vectorizations, cost models and solutions to the replication problem are examined and compared for all three languages. Based on the analysis a road map of a vectorization formulation and proof outline is sketched.

NESL, Proteus and Data Parallel Haskell all have a construct similar to the apply-to-each construct. The main issue of vectorization is how to remove this construct. All three language have a different approach to this. The level of detail in the treatment of cost varies greatly between the three languages, but they all address the replication problem.

## 2.1 NESL

**Vectorization** NESL is vectorized to an intermediate flat data parallel language called VCODE. In the process every apply-to-each construct is transformed to a VCODE expression that distributes the free variables over the new parallel degree and operations inside the apply-to-each are replaced by vector-version. If an operation has already been replaced by a vector-versions the vectorization inserts VCODE expressions that flattens the arguments and segmentates the result. This is in many ways equivalent to the naive vectorization.

**Cost Model** The NESL programming language comes with a built-in ideal cost model. It only reflects the source language cost model, and there is no comparison to a cost model of the flattened programs. There is a paper called "A Provable Time and Space Efficient Implementation of NESL", but it does not use flattening, so it is not interesting for this project.

**Replication Problem** The problem is present in the current version of NESL and the programmer is advised to hoist potentially expensive expression outside of apply-to-each constructs and to use a special operation for parallel indexing [Blelloch(1995), App C].

## 2.2 Proteus

**Vectorization** Proteus does not have an intermediate target language. Instead the target language is a subset of the source language. Every apply-to-each is pushed down to leaves of abstract syntax tree. For example

$$[f(x, y) : x \in v] \qquad \text{will translate to} \qquad f^{\mathrm{v}}([x : x \in v], [y : x \in v])$$

and

$$[f^{\mathrm{v}}(x, y) : x \in v] \qquad \text{will translate to} \qquad (f^{\mathrm{v}})^{\mathrm{v}}([x : x \in v], [y : x \in v]).$$

To eliminate operations of the form $(f^{\mathrm{v}})^{\mathrm{v}}$ flattening of the arguments and segmentation of the result is inserted as usual. In the leaves simple rules are applied such as

$$[x : x \in xs] \qquad \text{will translate to} \qquad xs$$

and

$$[y : x \in xs] \qquad \text{will translate to} \qquad \mathit{distribute}(1, \mathit{length}(xs)).$$

This style is roughly equivalent to naive vectorization.

**Cost Model** Proteus is the only language where a cost model is presented and the preservation of cost has been proved correct [Riely et al.(1995)Riely, Prins, and Iyer]. The ideal cost semantics is concluded to require a reference-based target language, and it is deemed impossible to implement. They present two other cost semantics that can be shown correct with resepct to both work and step complexity under certain circumstances. The two models are called construct-parameters semantics and construct-result semantics. Both models are more lenient than the ideal cost semantics and are therefore easier to implement. The construct-parameters semantics relaxes the cost of the apply-to-each construct by charging the full size of all the values bound to free variables in the expression. The construct-result semantics bases complexities on output size rarther than input size, but this leads to various undesirable restrictions on the programs. In all three cost models, the cost model is only given for the source language, and the cost model of the target language is then implied by constraints on the work-complexity of the vector-version of each primitive operation. They are able to do this because the source and target language is the same.

**Replication Problem** The replication problem has been solved for indexing by using special transformation rules. Potentially expensive expressions are hoisted out in the vectorization phase when possible [Keller and Simons(1996), Palmer et al.(1995)Palmer, Prins, and Westfold].

## 2.3 Data Parallel Haskell

**Vectorization** Data Parallel Haskell programs are simply Haskell programs that make use of Haskell libraries containing parallel vector-version of familiar Haskell functions. The apply-to-each construct and the vectorization phase is then built into the Glasgow Haskell Compiler [Chakravarty et al.(2007)Chakravarty, Leshchinskiy, Jones, Keller, and Marlow]. Data Parallel Haskell stands out from the two other language by being the only language that supports higher-order functions. This is also used when removing apply-to-each construct by desugaring them to a higher-order parallel function $map^{\mathrm{v}}$. The compiler then uses rewrite rules such as

$$map^{\mathrm{v}}\ f\ x \rightarrow f^{\mathrm{v}}\ x$$

and

$$(f^{\mathrm{v}})^{\mathrm{v}}\ x \rightarrow segment^{\mathrm{v}}\ x\ (f^{\mathrm{v}}\ (concat^{\mathrm{v}}\ x)).$$

**Cost Model** None of the literature on Data Parallel Haskell presents a formal cost model.

**Replication Problem** The replication problem regarding indexing is solved by using special deforestation (also called fusion) rules in Data Parallel Haskell [Chakravarty et al.(2007)Chakravarty, Leshchinskiy, Jones, Keller, and Marlow]. It is unclear if the deforestation is a general solution to the problem for all operations and user defined functions since the literature on the deforestation of vectorized programs is limited. The solution is likely comparable to Proteus'.

## 2.4 Comparison and Design Choices

The main emphasis on the presentation of vectorization in this report is the rigorousness of the proofs. As a consequence all unnecessary features and extensions of the presented languages are omited and only a small sufficient set of primitive operations is included. We also narrow the scope by leaving out a formal treatment of user defined functions. This seriously limits the expressiveness of the language, but as we show in section 4 the languages should be extendible with mutually recursive user-defined function without harming the work-efficiency of the vectorization.

**Vectorization** Higher-order functions are not included since they are difficult to parallelize [Roman Leshchinskiy and Keller(2006)], and as a result we cannot use the Data Parallel Haskell vectorization style. The choice between the NESL and the Proteus style of flattening has been rarther arbitrary as they are largely equivalent. The Proteus style is more compositional, but it requires several phases. We will pursue the NESL style of vectorziation for its simplicity and since it allows for optimization at the level of the apply-to-each expression, although we do not actually consider any such optimizations. Furthermore we will stress the distincion between the high-level source language and the low-level target language by separing them in two languages like NESL and VCODE.

In the source language the apply-to-each construct will be restricted to the simplified form $[e : x \in \mathbf{ix}(n)]$ where $\mathbf{ix}(n)$ is a special operation that generates a sequence of consecutive integers from 0 to $n-1$.

To decrease the size of the target language, there will be no scalar values in the language except when calculating the parallel degree. Scalar values in the source language outside any apply-to-each will be represented by a list with a single element. This means that the standard primitve operations are not needed - only the vector-versions. While this design choice decreases the size of the target language, it has the unfortunate side-effect that the representation of values deviates slightly from the representation given in the introduction. Since we need to be able to distinquish scalar values from flat vector, all values will be attached an additional segment descriptor compared to the previous representation. For example

$$d = [1]$$

is now a representation of 1 instead of [1], and

$$d = [1, 2, 3] \qquad s_0 = [3]$$

is now a representation of $[1, 2, 3]$ instead of $[[1, 2, 3]]$. This representation can then be used to represent values in parallel with each other. For example

$$d = [1, 2, 3]$$

is now a representation of 1 and 2 and 3 in parallel with each other, and

$$d = [1, 2, 3, 4] \qquad s_0 = [2, 2]$$

is now a representation of $[1, 2]$ and $[3, 4]$ in parallel. The idea that a target value represents possibly several source language values allows replication to be specified without introducing a new level of nesting. The replicated value is simply the same value, but with a replication of the top-most segment descriptor.

The main advantage of this is that we can define the type of a vector-version operation $f^{\mathrm{v}}$ to be the same as the type of $(f^{\mathrm{v}})^{\mathrm{v}}$. This in turn means that the flattening of the arguments and the segmentation of the result is no longer needed. Instead a segmentation step is required on the result of an apply-to-each.

**Cost Model** Proteus has the most detailed cost treatment for vectorization. In this report we will present a completely separate cost model for the source and target language rather than having the target language cost model implied by the source language cost model to spell out what the desired complexities are, and to address every operation. Similar to the Proteus cost model, we will relax the ideal cost semantics in order for the proof to go through.

**Replication Problem** All three languages addresses the replication problem by handling indexing explicitly. In Proteus and Data Parallel Haskell the vectorization phase does this automatically while it must be done manually by the programmer in NESL.

As pointed out by Prins et al. in [Riely et al.(1995)Riely, Prins, and Iyer], the ideal cost semantic requires a reference-based implementation of the target language: The expression $[y : x \in [1..n]]$ must create a list of $n$ $y$'s, and the ideal cost semantics dictates that it most do so in proportionally $n$ steps. Since the value bound to $y$ can have any size, a reference-based target language must be necessary.

Manual Chakravarty has presented a more reference-based representation of nested vectors [Chakravarty(2011)]. Instead of only having the segment lengths in the segment descriptors it is possible to also store the segment starts. In this way it is possible that two adjacent vector elements are placed in discontiguous segments in the flat data vector. An even further generalization is to allow different data vectors in the same nested vector and then point to a specific data vector in each segment. This form of describing segments is called scattered segments.

In this report we will explore the possibilities of obtaining a work-efficient vectorization using segment descriptors containing segment lengths and starts but not scattered segments. We will refer to this type of segment descriptors as generalized segment descriptors, and represented vectors containing generalized segment descriptors will be referred to as generalized values. Segment descriptors containing only lengths will be referred to as normalized segment descriptors. The term normalized is used because when only the lengths are present in the segment descriptors, the starts will have to be implicit, so normalized values can be said to be in a canonical form. Different generalized values on the other hand may represent the same nested vector. As an example

$$v = [[1], [1, 3]]$$

could be represented by

$$d = [1, 1, 3] \qquad s_0 = [0, 1] \qquad l_0 = [1, 2]$$

where $s_0$ is the starts and $l_0$ is the lengths. It could also be represented by

$$d = [1, 0, 0, 0, 1, 3] \qquad s_0 = [0, 4] \qquad l_0 = [1, 2]$$

or even

$$d = [1, 3] \qquad s_0 = [0, 0] \qquad l_0 = [1, 2]$$

The replication problem can then be solved by simply attaching a new segment descriptor to the replicated value that references the same segments multiple times. Consider the last representation of $v$. The following generalized value is a representation of $[v, v, v]$

$$d = [1, 3]$$

$$\begin{array}{ll} s_0 = [0,1] & l_0 = [1,2] \\ s_1 = [0,0,0] & l_1 = [2,2,2] \end{array}$$

This approach clearly has a work complexity in the order of the number of replicates required. Some care should be taken though - since generalized values may have many references to the same segment, not all architectures can support this approach efficiently. It depends on the actual cost of concurrent reads to the same location. This is also the case for the special handling of the index function in Proteus as noted in [Palmer et al.(1995)Palmer, Prins, and Westfold]. Here they propose a solution called node-extension where some data is purposely replicated without an actual need to replicate it to reduce the number of concurrent reads. This solution could have interest for the vectorization in this report, but we will leave it for future work.

## 2.5 Road Map

In section 3.2 a source language is defined. This language is the high-level nested data parallel language. A syntax is presented that defines all source language expressions **SExp** along with a denotational semantic $\mathcal{S}[\![\cdot]\!]$ that maps expressions to source language values **SVal**. A type system $\vdash_{\mathcal{S}}$ is also given that defines the type of all expression and values **STyp**. the semantic includes a work complexity semantic. The semantics reflects the desired computations of source language programs as the source language programs are not meant to be evaluated without vectorizing them first.

In section 3.3 a target language is defined. Similar to the source language the domains for expressions **TExp**, values **TVal** and types **TTyp** are defined along with a semantic function $\mathcal{T}[\![\cdot]\!]$ and type system $\vdash_{\mathcal{T}}$. The semantic function will evaluate expressions to a generalized type of values. To compare them to source language values a normalization function $|\cdot|$ is presented that transforms generalized values into values with a normal form **TValN**.

In section 3.4 a vectorization function $\langle\!\langle \cdot \rangle\!\rangle$ is given that transform source language expressions into target language expressions. A type transformation $\ll \cdot \gg$ and a value transformation $\langle \cdot \rangle$ is also given. The value transformation transforms source language values to target language values in normal form. The relationship between the different domains and transformations can be seen in figure 2.5. The vectorization is formally shown to preserve values and work complexity. In the figure the preservation corresponds to taking the lower path from **SExp** to **TValN** is the same as taking the upper path with some constant $K$ difference in work complexity. Furthermore $K$ is shown to depend only on the size of the expression.

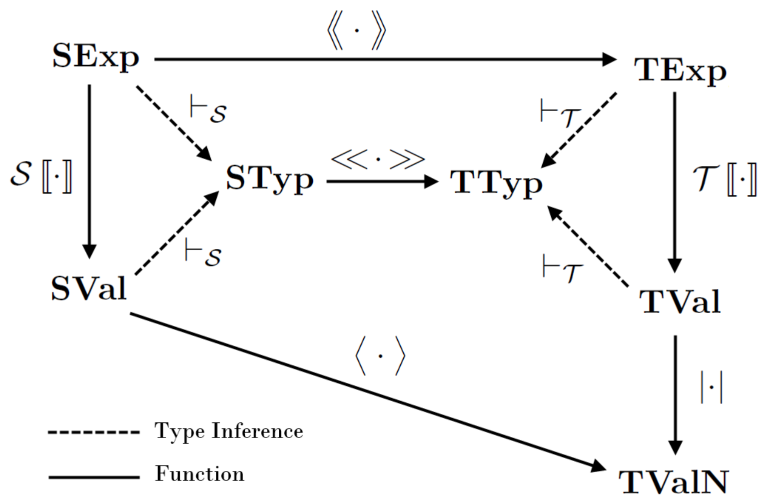



Figure 1: The relationship between the semantic function, type inference and transformations. Note that work complexity and error handling is not shown in this figure.

# 3 A Provable Work Efficient Vectorization

## 3.1 Mathematical Preliminaries

**Variable Names** $x$ will range over variable names **Var**.

$n$ will range over integers Z.

$k, i, j, l$ will range over natural numbers N. $i$ and $j$ will be used for indices, $k$ will be used to denote "small" numbers (related to program size), $l$ will be used for "large" numbers (related to data size).

**Numerals** Over-line is used to represent numerals, so $\overline{n}$ will denote the numeral representing the natural number $n$.

**Finite Maps** Finite maps are used both for variable environments and typing assumption contexts. The type of a finite map is written as

$$f : X \rightharpoonup_{\text{fin}} Y$$

Finite maps are written as

$$f = [x_1 \mapsto y_1, \ldots, x_k \mapsto y_k]$$

$dom(f)$ is the domain of a finite map $f$, so if $f = [x_1 \mapsto y_1, \ldots, x_k \mapsto y_k]$ then $dom(f) = \{x_1, \ldots, x_k\}$.

**Restriction** If $f$ is a domain of type $X \rightharpoonup_{\text{fin}} Y$ and $X' \subseteq X$ then

$$f \upharpoonright X'$$

will denote the restriction of $f$ to the variables in $X'$ such that

$$f \upharpoonright X'(x) = \begin{cases} f(x) & x \in X' \\ \mathit{undefined} & x \notin X' \end{cases}$$

**Inclusion** If $f$ and $f'$ are two finite maps of the same type then the inclusion of $f$ in $f'$ is defined as

$$f \subseteq f' \quad \textit{if and only if} \quad \forall x \in \mathit{dom}(f).f(x) = f'(x)$$

**Sets** $A^k$ will denote the sequence $\overbrace{A \times \cdots \times A}^{k}$.
$A^*$ will denote a finite sequence of $A$ of arbitrary size (possibly zero), so

$$A^* = \cup_{k \in \mathrm{N}} A^k$$

## 3.2 Source Language

### 3.2.1 Syntax

**Expressions**

$$\begin{array}{rcl} \mathbf{SExp} \ni e & ::= & \overline{r} \mid x \mid \mathbf{let}\ x \Leftarrow e_0\ \mathbf{in}\ e_1 \mid o(e_1, \ldots, e_k) \mid [e_0 : x \in \mathbf{ix}(e_1)] \\ r & ::= & n \mid \cdots \\ \mathbf{SOp} \ni o & ::= & \mathbf{vec}_k \mid \mathbf{length} \mid \mathbf{elt} \mid \mathbf{conc} \mid \mathbf{part} \mid \oplus \\ \oplus & ::= & + \mid <= \mid \mathbf{neg} \mid \cdots \end{array}$$

**Syntactic Sugar**

**Special Variables** Variable names beginning with an underscore (eg. $\_x$) are assumed to be selected such that they do not conflict with any other variables in the typing context. They are used for special variables inserted in the desugaring or vectorization phase.

**Vector Constructor**

$$[e_1, \ldots, e_k] \stackrel{\text{DEF}}{\equiv} \mathbf{vec}_k(e_1, \ldots, e_k) \quad (k \geq 0)$$

**Multiple Let-Bindings**

$$\begin{array}{ll} \mathbf{let} & x_1 \Leftarrow e_1 \\ & \vdots \\ & x_k \Leftarrow e_k \\ \mathbf{in} & e_0 \end{array} \quad \stackrel{\text{DEF}}{\equiv} \quad \begin{array}{llll} \mathbf{let} & x_1 \Leftarrow e_1 & \mathbf{in} & \\ \vdots & \vdots & \vdots & \\ \mathbf{let} & x_k \Leftarrow e_k & \mathbf{in} & e_0 \end{array}$$

**If-Then-Else**

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \stackrel{\text{DEF}}{\equiv} \begin{array}{ll} \textbf{let} & \_b \Leftarrow e_1 \\ \textbf{in} & \textbf{elt}([e_2 : \_ \in \textbf{ix}(\_b)] \mathbin{+\!\!+} [e_3 : \_ \in \textbf{ix}(\textbf{not}(\_b))], 0) \end{array}$$

**General Apply-To-Each**

$$[e_0 : x_1 \in e_1, \ldots, x_k \in e_k] \stackrel{\text{DEF}}{\equiv} \begin{array}{ll} \textbf{let} & \_x_1 \Leftarrow e_1 \\ & \vdots \\ & \_x_k \Leftarrow e_k \\ \textbf{in} & [e_0[\theta] : i \in \textbf{ix}(\textbf{length}(\_x_1))] \end{array} \qquad (k \geq 1)$$

where $\theta = [\textbf{elt}(\_x_1, i)/x_1, \ldots, \textbf{elt}(\_x_k, i)/x_k]$ is a substitution and expression substitution is defined in the obvious way.

This definition truncates the length of all vectors to the length of the first. A safer (and perhaps better) way to define the general apply-to-each construct would be to check that the length of all the iterator-bound values are the same.

**Standard Library Functions**

**Append**

$$e_0 \mathbin{+\!\!+} e_1 \stackrel{\text{DEF}}{\equiv} \textbf{conc}([e_0, e_1])$$

**Pack**

$$\textbf{pack}(e_0, e_1) \stackrel{\text{DEF}}{\equiv} \textbf{conc}([[\_x : \textbf{ix}(\_b)] : \_x \in e_0, \_b \in e_1])$$

### 3.2.2 Type System

**Domains**

$$\begin{array}{rcl} \textbf{STyp} \ni \tau & ::= & \pi \mid [\tau] \\ \textbf{TypPrim} \ni \pi & ::= & \text{int} \mid \cdots \end{array}$$

**Type Specialization** The type of a polymorphic primitive operation $o$ can be specialized by providing a type substitution for all the type variables. If an operation has the type

$$o : \forall \alpha_1, \ldots, \alpha_{k'} . \beta$$

where $\beta$ is of the form $\tau_1 * \ldots * \tau_k \;{-}{>}\; \tau \qquad (k \geq 0)$.

Then

$$o[\tau'_1, \ldots, \tau'_{k'}] : \beta[\tau'_1/\alpha_1, \ldots, \tau'_{k'}/\alpha_{k'}]$$

**Expressions** $\Gamma$ will range over typing assumptions contexts $\mathbf{Var} \rightharpoonup_{\text{fin}} \mathbf{STyp}$.

$\boxed{\Gamma \vdash_{\mathcal{S}} e : \tau}$

$$\frac{}{\Gamma \vdash_{\mathcal{S}} \overline{n} : \text{int}} \tag{SNum-T}$$

$$\frac{}{\Gamma \vdash_{\mathcal{S}} x : \tau}\ \Gamma(x) = \tau \tag{SVar-T}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_0 : \tau_0 \qquad \Gamma[x \mapsto \tau_0] \vdash_{\mathcal{S}} e_1 : \tau_1}{\Gamma \vdash_{\mathcal{S}} \textbf{let}\ x \Leftarrow e_0\ \textbf{in}\ e_1 : \tau_1} \tag{SLet-T}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash_{\mathcal{S}} e_k : \tau_k}{\Gamma \vdash_{\mathcal{S}} o(e_1, \ldots, e_k) : \tau}\ (o[\tau'_1, \ldots, \tau'_{k'}] : \tau_1 * \cdots * \tau_k \ {-}{>}\ \tau) \tag{SOp-T}$$

$$\frac{\Gamma \vdash_{\mathcal{S}} e_1 : \text{int} \qquad \Gamma[x \mapsto \text{int}] \vdash_{\mathcal{S}} e_0 : \tau}{\Gamma \vdash_{\mathcal{S}} [e_0 : x \in \mathbf{ix}(e_1)] : [\tau]} \tag{SApp-T}$$

**Operations**

$\mathbf{vec}_k : \forall\alpha.\alpha^k \ {-}{>}\ [\alpha] \quad \textit{for every } k \geq 0$

$\mathbf{length} : \forall\alpha.[\alpha] \ {-}{>}\ \text{int}$

$\mathbf{elt} : \forall\alpha.[\alpha] * \text{int} \ {-}{>}\ \alpha$

$\mathbf{conc} : \forall\alpha.[[\alpha]] \ {-}{>}\ [\alpha]$

$\mathbf{part} : \forall\alpha.[\alpha] * [\text{int}] \ {-}{>}\ [[\alpha]]$

All scalar operations types of the form

$\oplus : \pi_1 * \ldots * \pi_k \ {-}{>}\ \pi$

In particular

$\mathbf{+} : \text{int} * \text{int} \ {-}{>}\ \text{int}$

$<= : \text{int} * \text{int} \ {-}{>}\ \text{int}$

$\mathbf{neg} : \text{int} \ {-}{>}\ \text{int}$

**Properties**

**Lemma 1** *Typing Context Inclusion.*

$$\forall \Gamma, \Gamma'.$$

$$if\ \Gamma \subseteq \Gamma'\ and\ \Gamma \vdash_{\mathcal{S}} e : \tau\ then\ \Gamma' \vdash_{\mathcal{S}} e : \tau$$

**Proof** By simple induction on the derivations.

### 3.2.3 Denotational Semantics

**Computations** $w$ will range over work complexities **Work**.

The work of an evaluation of a source or target expression is measured in number of basic steps.

$$\mathbf{Work} = \mathrm{N}$$

The evaluation of expressions is encapsulated in a monad type:

$$\mathbf{M}^{\$}\ X = (X \times \mathbf{Work}) + \{\mathsf{E}\}$$

Values of type $\mathbf{M}^{\$}\ X$ are written as

$$\mathsf{R}_w\ x$$

where $x \in X$ and $w \in \mathbf{Work}$ and

$$\mathsf{E}$$

for the error value.

**Bind**

$$bind^{\$} : \mathbf{M}^{\$}\ X \to (X \to \mathbf{M}^{\$}\ Y) \to \mathbf{M}^{\$}\ Y$$

$$bind^{\$}\ (\mathsf{R}_{w_x}\ x)\ f = \begin{array}{l} \text{case } f(x) \text{ of} \\ \quad \begin{array}{ll} \mathsf{R}_{w_y}\ y \to & \mathsf{R}_{w_x + w_y}\ y \\ \mathsf{E} \quad\quad \to & \mathsf{E} \end{array} \end{array}$$

$$bind^{\$}\ \mathsf{E}\ f = \mathsf{E}$$

**Unit**

$$unit^{\$} : X \to \mathbf{M}^{\$}\ X$$

$$unit^{\$}\ x = \mathsf{R}_0\ x$$

**Tick**

$$\texttt{Tick} \;:\mathbf{Work} \to \mathbf{M}^{\$}\ 1$$

$$\texttt{Tick}_w \;= \mathsf{R}_w\ ()$$

**Let Notation**

$$let^{\$}\ x \leftarrow e_1\ in\ e_2 \overset{\text{DEF}}{\equiv} bind^{\$} e_1\ (\lambda x.e_2)$$

$$let^{\$}\ x = e_1\ in\ e_2 \overset{\text{DEF}}{\equiv} let^{\$}\ x \leftarrow unit^{\$}\ e_1\ in\ e_2$$

The $let^{\$}$ notation generalized to multi-line in the obvious way.

**Variable Notation** If $x$ ranges over $X$ then $x^{\$}$ will range over decorated values $\mathbf{M}^{\$}\ X$.

**Values**

$$\mathbf{SVal} \ni v ::= n \mid [v_1, \ldots, v_l]$$

**Value Types** $\boxed{\vdash_{\mathcal{S}} v : \tau}$

$$\frac{}{\vdash_{\mathcal{S}} n : \text{int}} \qquad \text{(SNat-T)}$$

$$\frac{\vdash_{\mathcal{S}} v_1 : \tau \quad \cdots \quad \vdash_{\mathcal{S}} v_l : \tau}{\vdash_{\mathcal{S}} [v_1, \ldots, v_l] : [\tau]}\ (l \geq 0) \qquad \text{(SVec-T)}$$

$\boxed{\vdash_{\mathcal{S}} \rho : \Gamma}$

$$\frac{\vdash_{\mathcal{S}} v_1 : \tau_1 \quad \cdots \quad \vdash_{\mathcal{S}} v_k : \tau_k}{\vdash_{\mathcal{S}} [x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] : [x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_k]} \qquad \text{(SEnv-T)}$$

$$\frac{\vdash_{\mathcal{S}} \rho : \Gamma \qquad \vdash_{\mathcal{S}} v : \tau}{\vdash_{\mathcal{S}} \rho[x \mapsto v] : \Gamma[x \mapsto \tau]} \qquad \text{(SEnvUp-T)}$$

**Auxiliary Functions**

**Length and Size**

$$\# : \mathbf{SVal} \rightharpoonup \mathrm{N} \qquad \|\cdot\| : \mathbf{SVal} \to \mathrm{N}$$

$$\#[v_1, \ldots, v_l] = l$$

$$\|v\| = \begin{cases} 1 + \sum_{i=1}^{l} \|v_i\| & v = [v_1, \ldots, v_l] \\ 1 & v = r \end{cases}$$

**Concat**

$$\wedge : \mathbf{SVal} \times \mathbf{SVal} \rightharpoonup \mathbf{SVal}$$

$$[v_1, \ldots, v_l] \wedge [v'_1, \ldots, v'_{l'}] = [v_1, \ldots, v_l, v'_1, \ldots, v'_{l'}]$$

**Scan**

$$scan_+ : \mathbf{SVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{SVal}$$

$$\begin{array}{lll} scan_+([n_1, \ldots, n_l]) = & \text{let} & n'_i = \sum_{i=1}^{l-1} n_i \qquad \forall i \in \{1..l+1\} \\ & \text{in} & \mathsf{R}_l\ [n'_1, \ldots n'_l] \end{array}$$

**Fetch**

$$\begin{array}{ll} fetch : \mathbf{SVal} \times \mathrm{N} \times \mathrm{N} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{SVal} & \\ fetch([v_0, \ldots, v_{l-1}], n, n') = & \textit{if}\ n \geq 0\ \textit{and}\ n' \geq 0\ \textit{and}\ n + n' < l \\ & \textit{then}\ \mathsf{R}_{1+n'}\ [v_n, \ldots, v_{n+n'-1}] \\ & \textit{else}\ \mathsf{E} \end{array}$$

**Expressions** $\rho$ will range over value environments $\mathbf{Var} \rightharpoonup_{\text{fin}} \mathbf{SVal}$. The expression evaluation function has the type

$$\mathcal{S} : \mathbf{SExp} \times (\mathbf{Var} \rightharpoonup_{\text{fin}} \mathbf{SVal}) \rightharpoonup \mathbf{M}^{\$}\ \mathbf{SVal}$$

It is only undefined for static errors such as unbound variable reference or type error.

$$\mathcal{O} : \mathbf{SOp} \times \mathbf{SExp}^* \rightharpoonup \mathbf{M}^{\$}\ \mathbf{SVal}$$

$$\boxed{\mathcal{S}\,[\![e]\!]\,\rho = v^{\$}}$$

$$\mathcal{S}\,[\![\overline{r}]\!]\,\rho \quad = \quad \mathsf{R}_1\ r$$

$$\mathcal{S}\,[\![x]\!]\,\rho \quad = \quad \mathsf{R}_1\ \rho(x)$$

$$\mathcal{S}\,[\![\mathbf{let}\ x \Leftarrow e_1\ \mathbf{in}\ e_2]\!]\,\rho \quad = \quad \begin{array}{ll} \mathit{let}^{\$} & v_1 \leftarrow \mathcal{S}\,[\![e_1]\!]\,\rho \\ \mathit{in} & \mathcal{S}\,[\![e_2]\!]\,\rho[x \mapsto v_1] \end{array}$$

$$\mathcal{S}\,[\![o(e_1,\ldots,e_k)]\!]\,\rho \quad = \quad \begin{array}{ll} \mathit{let}^{\$} & v_1 \leftarrow \mathcal{S}\,[\![e_1]\!]\,\rho \\ & \vdots \\ & v_k \leftarrow \mathcal{S}\,[\![e_k]\!]\,\rho \\ \mathit{in} & \mathcal{O}\,[\![o]\!]\,(v_1,\ldots,v_k) \end{array}$$

$$\mathcal{S}\,[\![[e_0 : x \in \mathbf{ix}(e_1)]]\!]\,\rho \quad = \quad \begin{array}{lll} \mathit{let}^{\$} & n \leftarrow \mathcal{S}\,[\![e_1]\!]\,\rho & \\ \mathit{in} & \mathit{if}\ \ n \geq 0 & \\ & \mathit{then} & v_1 \leftarrow \mathcal{S}\,[\![e_0]\!]\,\rho[x \mapsto 0] \\ & & \vdots \\ & & v_n \leftarrow \mathcal{S}\,[\![e_0]\!]\,\rho[x \mapsto n-1] \\ & & \mathsf{R}_{n+1}\ [v_1,\ldots,v_n] \\ & \mathit{else} & \mathsf{E} \end{array}$$

**Operations** $\boxed{\mathcal{O}\,[\![o]\!]\,(v_1,\ldots,v_k) = v^{\$}}$

$$\mathcal{O}\,[\![\mathbf{vec}_k]\!]\,(v_1,\ldots,v_k) \quad = \quad \mathsf{R}_{\sum_{i=1}^{k} \|v_i\|}\ [v_1,\ldots,v_k]$$

$$\mathcal{O}\,[\![\mathbf{length}]\!]\,(v) \quad = \quad \mathsf{R}_1\ \#v$$

$$\mathcal{O}\,[\![\mathbf{elt}]\!]\,([v_0,\ldots,v_{l-1}],n) \quad = \quad \begin{array}{ll} \mathit{if} & 0 \leq n \leq l-1 \\ \mathit{then} & \mathsf{R}_1\ v_n \\ \mathit{else} & \mathsf{E} \end{array}$$

$$\mathcal{O}\,[\![\mathbf{conc}]\!]\,([v_1,\ldots,v_l]) \quad = \quad \mathsf{R}_{l+\sum_{i=1}^{l} \#v_i}\ v_1 \mathbin{\hat{}} \ldots \mathbin{\hat{}} v_l$$

$$\mathcal{O}\,[\![\mathbf{part}]\!]\,(v,[n_1,\ldots,n_l]) \quad = \quad \begin{array}{ll} \mathit{let}^{\$} & [n'_1,\ldots,n'_l] \leftarrow \mathit{scan}_+([n_1,\ldots,n_l]) \\ & v'_1 \leftarrow \mathit{fetch}(v,n'_1,n_1) \\ & \vdots \\ & v'_l \leftarrow \mathit{fetch}(v,n'_l,n_l) \\ \mathit{in} & \mathsf{R}_l\ [v'_1,\ldots,v'_l] \end{array}$$

$$\mathcal{O}\,[\![\oplus]\!]\,(r_1,\ldots,r_k) \quad = \quad \mathsf{R}_1\ \oplus(r_1,\ldots,r_k)$$

### 3.2.4 Properties

**Lemma 2** *Type Soundness.*

$$
\begin{array}{lll}
\textit{if } \Gamma \vdash_{\mathcal{S}} e : \tau \textit{ and } \vdash_{\mathcal{S}} \rho : \Gamma & & \\
\textit{then} & \textit{either} & \mathcal{S}\,[\![e]\!]\,\rho = \mathsf{E} \\
& \textit{or} & \mathcal{S}\,[\![e]\!]\,\rho = \mathsf{R}_w\ v \quad \textit{and} \quad \vdash_{\mathcal{S}} v : \tau
\end{array}
$$

**Proof** By induction on $e$.

Type soundness tells us that the semantic function is total for well-typed expressions and environments.

## 3.3 Target Language

### 3.3.1 Syntax

**Expressions** Target language expressions are divided into two syntactic categories to simplify notation: "Serious" expressions $s \in \mathbf{TExp}_\mathrm{s}$ and "trivial" expression $t \in \mathbf{TExp}_\mathrm{t}$. Trivial expressions are expressions that have constant time work complexity, and can never fail in a type-correct program. This allows the evaluation function of trivial expressions to be non-monadic. Serious expressions on the other hand can be any expression including trivial expressions, so they have a non-trivial work complexity, and can fail due to runtime errors such as indexing out of bounds.

$$
\begin{array}{rcl}
\mathbf{TExp}_\mathrm{t} \ni t & ::= & x \mid \mathbf{attsd}(t_1, t_2) \mid \mathbf{data}(t) \mid \mathbf{seg}(t) \mid \mathbf{starts}(t) \mid \mathbf{lens}(t) \\
\mathbf{TExp}_\mathrm{s} \ni s & ::= & t \mid [\overline{r}] \mid \mathbf{let}\ x \Leftarrow s_0\ \mathbf{in}\ s_1 \mid p(t_1, \ldots, t_k) \mid \mathbf{par}\ u\ \mathbf{do}\ s \\
\mathbf{TScalarExp} \ni u & ::= & \mathbf{length}(t) \\
\mathbf{TOp} \ni p & ::= & \mathbf{mkseg} \mid \mathbf{segsum} \mid \mathbf{segfetch} \mid \mathbf{pd} \mid \mathbf{zip}_k \mid \mathbf{ixrep} \mid \mathbf{iotas} \mid \oplus^\mathrm{v} \\
\oplus^\mathrm{v} & ::= & +^\mathrm{v} \mid <=^\mathrm{v} \mid \mathbf{neg}^\mathrm{v} \mid \cdots
\end{array}
$$

**Syntactic Sugar**

$$
\begin{array}{ll}
\mathbf{let} & x_1 \Leftarrow s_1 \\
& \vdots \\
& x_k \Leftarrow s_k \\
\mathbf{in} & s_0
\end{array}
\quad \overset{\mathrm{DEF}}{\equiv} \quad
\begin{array}{lll}
\mathbf{let} & x_1 \Leftarrow s_1 & \mathbf{in} \\
\vdots & \vdots & \vdots \\
\mathbf{let} & x_k \Leftarrow s_k & \mathbf{in}\ s_0
\end{array}
$$

### 3.3.2 Type System

**Domains** $\sigma$ will range over types **TTyp**.

$\dot{\sigma}$ will range over generalized types **TTypGen** $\subset$ **TTyp**.

$\hat{\sigma}$ will range over normalized types **TTypNorm** $\subset$ **TTyp**.

$$\sigma ::= [\pi] \mid (\sigma, \sigma)$$

Generalized types:

$$\text{seg} \stackrel{\text{DEF}}{\equiv} ([\text{int}], [\text{int}])$$

$$\dot{\sigma} ::= [\pi] \mid (\text{seg}, \dot{\sigma})$$

Normalized types:

$$\hat{\sigma} ::= [\pi] \mid ([\text{int}], \hat{\sigma})$$

**Expressions**

$\Pi$ will range over typing assumptions environments **Var** $\rightharpoonup_{\text{fin}}$ **TTyp**.

$\boxed{\Pi \vdash_{\mathcal{U}} u}$

$$\frac{\Pi \vdash_{\mathcal{T}_t} t : [\pi]}{\Pi \vdash_{\mathcal{U}} \mathbf{length}(t)} \qquad \text{(U-T)}$$

$\boxed{\Pi \vdash_{\mathcal{T}_t} t : \sigma}$

$$\frac{}{\Pi \vdash_{\mathcal{T}_t} x : \sigma}\ \Pi(x) = \sigma \qquad \text{(TVAR-T)}$$

$$\frac{\Pi \vdash_{\mathcal{T}_t} t_1 : \text{seg} \quad \Pi \vdash_{\mathcal{T}_t} t_2 : \sigma}{\Pi \vdash_{\mathcal{T}_t} \mathbf{attsd}(t_1, t_2) : (\text{seg}, \sigma)} \qquad \text{(TATTSD-T)}$$

$$\frac{\Pi \vdash_{\mathcal{T}_t} t : (\text{seg}, \sigma)}{\Pi \vdash_{\mathcal{T}_t} \mathbf{data}(t) : \sigma} \qquad \text{(TDATA-T)}$$

$$\frac{\Pi \vdash_{\mathcal{T}_t} t : (\text{seg}, \sigma)}{\Pi \vdash_{\mathcal{T}_t} \mathbf{seg}(t) : \text{seg}} \qquad \text{(TSEG-T)}$$

$$\frac{\Pi \vdash_{\mathcal{T}_t} t : \text{seg}}{\Pi \vdash_{\mathcal{T}_t} \mathbf{starts}(t) : [\text{int}]} \qquad \text{(TSTARTS-T)}$$

$$\frac{\Pi \vdash_{\mathcal{T}_t} t : \mathrm{seg}}{\Pi \vdash_{\mathcal{T}_t} \mathbf{lens}(t) : [\mathrm{int}]} \tag{TLens-T}$$

$\boxed{\Pi \vdash_{\mathcal{T}_s} s : \sigma}$

$$\frac{\Pi \vdash_{\mathcal{T}_t} t : \sigma}{\Pi \vdash_{\mathcal{T}_s} t : \sigma} \tag{Trivial-T}$$

$$\frac{}{\Pi \vdash_{\mathcal{T}_s} [\overline{n}] : [\mathrm{int}]} \tag{TNumV-T}$$

$$\frac{\Pi \vdash_{\mathcal{T}_s} s_0 : \sigma_0 \qquad \Pi[x \mapsto \sigma_0] \vdash_{\mathcal{T}_s} s_1 : \sigma_1}{\Pi \vdash_{\mathcal{T}_s} \mathbf{let}\ x \Leftarrow s_0\ \mathbf{in}\ s_1 : \sigma_1} \tag{TLet-T}$$

$$\frac{\Pi \vdash_{\mathcal{T}_s} t_1 : \sigma_1 \qquad \cdots \qquad \Pi \vdash_{\mathcal{T}_s} t_k : \sigma_k}{\Pi \vdash_{\mathcal{T}_s} p(t_1, \ldots, t_k) : \sigma} \, (p[\sigma'_1, \ldots, \sigma'_{k'}] : \sigma_1 * \cdots * \sigma_k \texttt{ -> } \sigma) \tag{TOp-T}$$

$$\frac{\Pi \vdash_{\mathcal{U}} u \quad \Pi \vdash_{\mathcal{T}_s} s : \sigma}{\Pi \vdash_{\mathcal{T}_s} \mathbf{par}\ u\ \mathbf{do}\ s : \sigma} \tag{TPar-T}$$

**Operations**

$\mathbf{mkseg} : [\mathrm{int}] \texttt{ -> } \mathrm{seg}$

$\mathbf{segsum} : [\mathrm{int}] * [\mathrm{int}] \texttt{ -> } [\mathrm{int}]$

$\mathbf{ixrep} : [\mathrm{int}] \texttt{ -> } [\mathrm{int}]$

$\mathbf{iotas} : [\mathrm{int}] \texttt{ -> } [\mathrm{int}]$

$\mathbf{zip}_k : \forall \alpha . \alpha^k \texttt{ -> } \alpha$

$\mathbf{segfetch} : \forall \alpha . \alpha * [\mathrm{int}] * [\mathrm{int}] \texttt{ -> } \alpha$

All vector-versions of scalar operations have types of the form

$\oplus^{\mathrm{v}} : [\pi_1] * \ldots * [\pi_k] \texttt{ -> } [\pi]$

In particular

$+^{\mathrm{v}} : [\mathrm{int}] * [\mathrm{int}] \texttt{ -> } [\mathrm{int}]$

$\texttt{<=}^{\mathrm{v}} : [\mathrm{int}] * [\mathrm{int}] \texttt{ -> } [\mathrm{int}]$

$\mathbf{neg}^{\mathrm{v}} : [\mathrm{int}] \texttt{ -> } [\mathrm{int}]$

**Properties**

**Lemma 3** *Type Context Inclusion.*

$\forall \Pi, \Pi'.$

$if\ \Pi \subseteq \Pi'\ and\ \Pi \vdash_{\mathcal{T}_s} s : \sigma\ then\ \Pi' \vdash_{\mathcal{T}_s} s : \sigma$

**Proof** By simple induction on the derivations with a separate lemma for trivial expression types.

### 3.3.3 Denotational Semantics

**Values** The semantics of the target language will be based on the generalized representation of values. This means that the operation **segfetch** can work on the top-most segment descriptor. This is the key to handling the replication problem in general.

$c, \overline{c}, \hat{c}$ will range over values **TVal**.

Where $\overline{c}$ will be used for flat vectors only (including the empty vector), $\hat{c}$ will be used for normalized values only and $c$ will be used for any value (mostly generalized values). Many operations would normally require the empty vector to be fully constructed meaning that even though the vector is empty, it still has the correct number of segment descriptors (although empty) as prescribed by its type. In order to avoid having to type annotate all polymorphic operations in the target language, we introduce a special empty vector value `[]`, and each primitive operation is then defined with a special case for `[]` (if necessary).

$$
\begin{array}{rcl}
\mathbf{TVal} \ni c & ::= & \overline{c} \mid (c, c) \\
\overline{c} & ::= & [r_1, \ldots, r_l] \mid \texttt{[]} \\
\mathbf{TValN} \ni \hat{c} & ::= & \overline{c} \mid ([\mathrm{int}], \hat{c})
\end{array}
$$

**Value Types** $\boxed{\vdash_{\mathcal{T}_s} c : \sigma}$

$$\frac{}{\vdash_{\mathcal{T}_s} [n_1, \ldots, n_l] : [\mathrm{int}]} \quad (l > 0) \qquad \text{(TNatV-T)}$$

$$\frac{}{\vdash_{\mathcal{T}_s} \texttt{[]} : \sigma} \qquad \text{(Empty-T)}$$

$$\frac{\vdash_{\mathcal{T}_s} c_1 : \sigma_1 \qquad \vdash_{\mathcal{T}_s} c_2 : \sigma_2}{\vdash_{\mathcal{T}_s} (c_1, c_2) : (\sigma_1, \sigma_2)} \qquad \text{(Pair-T)}$$

$\boxed{\vdash_{\mathcal{S}} \phi : \Pi}$

$$\frac{\vdash_{\mathcal{T}_s} c_1 : \sigma_1 \quad \cdots \quad \vdash_{\mathcal{T}_s} c_k : \sigma_k}{\vdash_{\mathcal{T}} [x_1 \mapsto c_1, \ldots, x_k \mapsto c_k] : [x_1 \mapsto \sigma_1, \ldots, x_k \mapsto \sigma_k]} \quad (\text{TE}{\small NV}\text{-T})$$

$$\frac{\vdash_{\mathcal{T}} \phi : \Pi \qquad\qquad \vdash_{\mathcal{T}_s} c : \sigma}{\vdash_{\mathcal{T}} \phi[x \mapsto c] : \Pi[x \mapsto \sigma]} \quad (\text{TE}{\small NV}\text{U}{\small P}\text{-T})$$

**Auxiliary Functions**

**First and Second Projection**

$fst : \mathbf{TVal} \rightharpoonup \mathbf{TVal}$

$$fst(c) = \begin{cases} c_1 & (c = (c_1, c_2)) \\ \texttt{[]} & (c = \texttt{[]}) \end{cases}$$

$snd : \mathbf{TVal} \rightharpoonup \mathbf{TVal}$

$$snd(c) = \begin{cases} c_2 & (c = (c_1, c_2)) \\ \texttt{[]} & (c = \texttt{[]}) \end{cases}$$

**Length**

$\# : \mathbf{TVal} \rightharpoonup \mathrm{N}$

$\#[r_1, \ldots, r_l] = l$

$\#((\bar{c}_1, \bar{c}_2), c) = \#\bar{c}_1$

$\#\texttt{[]} = 0$

**Size** We only consider the size of normalized values, since the size of generalized values may be arbitrary large compared to the source value they represent.

$\|\cdot\| : \mathbf{TValN} \to \mathrm{N}$

$\|[r_1, \ldots, r_l]\| = 1 + l$

$\|(\bar{c}_1, \bar{c}_2)\| = 1 + \#\bar{c}_1 + \|\bar{c}_2\|$

$\|\texttt{[]}\| = 1$

**Depth**

$$\delta : \mathbf{TVal} \to \mathrm{N}$$

$$\delta([r_1, \ldots, r_l]) = 0$$

$$\delta((c_1, c_2)) = 1 + \delta(c_2)$$

$$\delta(\texttt{[]}) = 0$$

**Concatenation** When multiple vectors need to be concatenated, the cost of using a binary concat operator would be quadratic. It is however well-known that it is possible to implement k-ary concatenation in linear time.

$$conc_k : \mathbf{TVal}^k \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$$conc_k(\bar{c}_1, \ldots, \bar{c}_k) = \mathsf{R}_{k+\sum_{i=1}^{k} \#\bar{c}_i}\ \bar{c}_1 \wedge \cdots \wedge \bar{c}_k$$

where

$$\wedge : \mathbf{TVal} \times \mathbf{TVal} \rightharpoonup \mathbf{TVal}$$

$$[r_1, \ldots, r_l] \wedge [r'_1, \ldots, r'_{l'}] = [r_1, \ldots, r_l, r'_1, \ldots, r'_{l'}]$$

$$\texttt{[]} \wedge \texttt{[]} = \texttt{[]}$$

$$\texttt{[]} \wedge \bar{c} = \bar{c} \wedge \texttt{[]} = \bar{c}$$

**Replication**

$$[\cdot]. : \mathrm{Z} \times \mathrm{N} \to \mathbf{TVal}$$

$$0 \bullet r = \texttt{[]}$$

$$l \bullet r = [\overbrace{r, \ldots, r}^{l}] \quad (l > 0)$$

**Index Replication**

$$ixrep_k : \mathrm{N}^k \to \mathbf{TVal}$$

$$ixrep_k(n_1, \ldots, n_k) = n_1 \bullet 0 \wedge \cdots \wedge n_k \bullet k - 1$$

**Iotas**

$$iotas_k : \mathrm{N}^k \to \mathbf{TVal}$$

$$iotas_k(n_1, \ldots, n_k) = \iota(n_1) \wedge \cdots \wedge \iota(n_k)$$

where

$$\iota : \mathrm{N} \to \mathbf{TVal}$$

$$\iota(n) = [0, \ldots, n-1]$$

**Scan**

$$scan_+ : \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$$scan_+([n_1,\ldots,n_l]) = \begin{array}{ll} \text{let} & n'_i = \sum_{i=1}^{l-1} n_i \qquad \forall i \in \{1..l+1\} \\ \text{in} & \mathsf{R}_l\ [n'_1,\ldots n'_l] \end{array}$$

$$scan_+(\texttt{[]}) = \mathsf{R}_1\ \texttt{[]}$$

**Fetch**

$$\begin{array}{ll} fetch : \mathbf{TVal} \times \mathrm{N} \times \mathrm{N} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal} & \\ fetch([r_0,\ldots,r_{l-1}],n,n') = & if\ n \geq 0\ and\ n' \geq 0\ and\ n+n' < l \\ & then\ \mathsf{R}_{1+n'}\ [r_n,\ldots,r_{n+n'-1}] \\ & else\ \mathsf{E} \end{array}$$

$$fetch(\overline{c},l,0) = \mathsf{R}_1\ \texttt{[]} \qquad (l < \#\overline{c})$$

**Partition**

$$part : \mathbf{TVal} \times \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}^*$$

$$part([r_1,\ldots,r_l],[n_1,\ldots,n_{l'}])$$

$$\begin{array}{llll} = & if & \sum_{i=1}^{l'} n_i = l & \\ & then & let^{\$} & [n'_1,\ldots,n'_{l'}] \leftarrow scan_+([n_1,\ldots,n_{l'}]) \\ & & in & \mathsf{R}_l\ ([r_{n'_1},\ldots,r_{n'_2-1}],\ldots,[r_{n'_{l'}},\ldots,r_l]) \\ & else & \mathsf{E} & \end{array}$$

$$part(\texttt{[]},\overbrace{[0,\ldots,0]}^{l}) = \mathsf{R}_l\ (\overbrace{\texttt{[]},\ldots,\texttt{[]}}^{l}) \qquad (l > 0)$$

Work complexity: If $\#\overline{c}_1 = l$ and $\#\overline{c}_2 = l'$ then the work complexity of $part(\overline{c}_1,\overline{c}_2)$ is $l + l'$.

**Segmented Sum**

$$segsum : \mathbf{TVal} \times \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$$segsum(\overline{c}_1,\overline{c}_2) = \begin{array}{ll} let^{\$} & (\overline{c}'_1,\ldots,\overline{c}'_l) \leftarrow part(\overline{c}_1,\overline{c}_2) \\ & \overline{c}''_1 \leftarrow sum(\overline{c}'_1) \\ & \vdots \\ & \overline{c}''_l \leftarrow sum(\overline{c}'_l) \\ in & \mathsf{R}_l\ [\overline{c}''_1,\ldots,\overline{c}''_l] \end{array}$$

where

$$sum : \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathrm{N}$$

$$sum([n_1,\ldots,n_l]) = \mathsf{R}_l \sum_{i=1}^{l} n_i$$

$$sum(\texttt{[]}) = \mathsf{R}_1\ 0$$

Work complexity: If $\#\overline{c}_1 = l$ and $\#\overline{c}_2 = l'$ then the work complexity of $segsum(\overline{c}_1, \overline{c}_2)$ is $2(l + l')$.

**Type Predicates** To simplify pattern matching notation the following type predicates are introduced.

$$flat? : \mathbf{TVal} \to \mathrm{B} \qquad nested? : \mathbf{TVal} \to \mathrm{B} \qquad empty? : \mathbf{TVal} \to \mathrm{B}$$

$$empty?\ c = c\ is\ of\ the\ form\ \texttt{[]}$$

$$flat?\ c = c\ is\ of\ the\ form\ [n_1,\ldots,n_l]$$

$$pair?\ c = c\ is\ of\ the\ form\ (c_1, c_2)$$

**Make Starts** Make starts creates a generalized segment descriptor from a list of segments lengths assuming the data it refers to is aligned as in a normalized value.

$$makeStarts : \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$$makeStarts(\overline{c}) = \begin{array}{ll} let^{\$} & \overline{c}' \leftarrow scan_{+}(\overline{c}) \\ in & \mathsf{R}_1\ (\overline{c}', \overline{c}) \end{array}$$

**Segmented Fetch** The *segfetch* function fetches values from a target language value given a list of start indices and fetching lengths, and it is primarily used for the **segfetch** operation. The function can fetch from values of any nesting depth. The operation is strictly speaking redundant and could be replaced by two primitive operations **fetch** and $\mathbf{conc}_k$, but this would require the vectorization phase to produce different code for flat and nested values, so this solution allows a more simple type-insensitive vectorization. In a real implementation it would probably be easier and better to do the type checking in the vectorization phase, but when reasoning about the vectorization this approach is simpler. Generalized segment descriptors allows *segfetch* to only operate on the top-most level of segment descriptors. Fetching from a value with huge segments is therefore cheap (eg. replication of large vectors).

$$segfetch : \mathbf{TVal} \times \mathbf{TVal} \times \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$$segfetch(c, \overline{c}_1, \overline{c}_2) = \begin{cases} sfFlat(c, \overline{c}_1, \overline{c}_2) & (flat?\ c) \\ sfNested(c, \overline{c}_1, \overline{c}_2) & (pair?\ c) \\ \mathsf{R}_1\ \texttt{[]} & (empty?\ c \wedge (\overline{c}_2 = [0,\ldots,0])) \end{cases}$$

$$sfFlat : \mathbf{TVal} \times \mathbf{TVal} \times \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$$\begin{array}{lll} & \multicolumn{2}{l}{sfFlat(\overline{c}, [n_1, \ldots, n_l], [n'_1, \ldots, n'_l])} \\ = & let^{\$} & \overline{c}'_1 \leftarrow fetch(\overline{c}, n_1, n'_1) \\ & & \vdots \\ & & \overline{c}'_l \leftarrow fetch(\overline{c}, n_l, n'_l) \\ & in & conc_l(\overline{c}'_1, \ldots, \overline{c}'_l) \end{array}$$

$$sfNested : \mathbf{TVal} \times \mathbf{TVal} \times \mathbf{TVal} \rightharpoonup \mathbf{TVal}$$

$$\begin{array}{lll} sfNested(((\overline{c}_1, \overline{c}_2), c), \overline{c}_3, \overline{c}_4) = & let^{\$} & \overline{c}'_1 \leftarrow sfFlat(\overline{c}_1, \overline{c}_3, \overline{c}_4) \\ & & \overline{c}'_2 \leftarrow sfFlat(\overline{c}_2, \overline{c}_3, \overline{c}_4) \\ & in & \mathsf{R}_1\ ((\overline{c}'_1, \overline{c}'_2), c) \end{array}$$

Notice that *sfNested* does not recurse on the structure of the argument. It simply changes the top most segment descriptor.

**Work Complexity** Let $[n_1, \ldots, n_l]$ be the third argument to *segfetch* and let $l' = \sum n_i$. Then *sfFlat* costs $2l + 2l'$ and *sfNested* costs $1 + 4l + 4l'$. So the work complexity of *segfetch* is no more than $1 + 4l + 4l'$.

**Segmented Zip** Segmented zip is used to implement the vector constructor. Segmented zip takes $2k$ arguments. The first $k$ are values of the same nesting depth and the last $k$ are vectors containing segment lengths. The function combines the values by interleaving the elements of the values according to the given segmentation. Conceptually

$$segzip_2([a, b, c, d], [x, y], [2, 2], [1, 1]) = [a, b, x, c, d, y]$$

The *segzip* function is only defined on normalized values. This means that arguments will have to undergo expensive normalization before *segzip* can be used. Furthermore the result type is normalized so the return value will have to be denormalized to be used any further. Like the **segfetch** operation the $\mathbf{zip}_k$ operation could be replaced by primitive **part**, **segsum** and $\mathbf{conc}_k$ operations, but this would require type information to be available in the vectorization phase.

$$segzip_k : \mathbf{TValN}^k \times \mathbf{TVal}^k \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TValN}$$

$$segzip_k(\hat{c}_1, \ldots, \hat{c}_k, \overline{c}_1, \ldots, \overline{c}_k) = \begin{cases} szFlat_k(\hat{c}_1, \ldots, \hat{c}_k, \overline{c}_1, \ldots, \overline{c}_k) & (\forall i. flat?\ \hat{c}_i \vee empty?\ \hat{c}_i) \\ szNested_k(\hat{c}_1, \ldots, \hat{c}_k, \overline{c}_1, \ldots, \overline{c}_k) & (\exists i. pair?\ \hat{c}_i) \end{cases}$$

Note that because of the existential quantification in the second case of $segzip_k$, the second case will be false if $k = 0$. On the other hand the first case will not, so $segzip_0() = sfFlat_0()$.

$$szFlat_k : \mathbf{TValN}^k \times \mathbf{TVal}^k \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TValN}$$

$$\begin{array}{lll} szFlat_k(\overline{c}_1, \ldots, \overline{c}_k, \overline{c}'_1, \ldots, \overline{c}'_k) = & let^{\$} & (\overline{c}''_{1,1}, \ldots, \overline{c}''_{1,k'}) \leftarrow part(\overline{c}_1, \overline{c}'_1) \qquad (\#\overline{c}'_1 = k') \\ & & \qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\ & & (\overline{c}''_{k,1}, \ldots, \overline{c}''_{k,k'}) \leftarrow part(\overline{c}_k, \overline{c}'_k) \qquad (\#\overline{c}'_k = k') \\ & & \overline{c}'''_1 \leftarrow conc_k(\overline{c}''_{1,1}, \ldots, \overline{c}''_{k,1}) \\ & & \ \vdots \qquad\qquad \vdots \\ & & \overline{c}'''_{k'} \leftarrow conc_k(\overline{c}''_{1,k'}, \ldots, \overline{c}''_{k,k'}) \\ & in & conc_{k'}(\overline{c}'''_1, \ldots, \overline{c}'''_{k'}) \end{array}$$

$$szNested_k : \mathbf{TValN}^k \times \mathbf{TVal}^k \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TValN}$$

$$\begin{array}{lll} szNested_k(\hat{c}_1, \ldots, \hat{c}_k, \overline{c}_1, \ldots, \overline{c}_k) = & let^{\$} & \overline{c}'_1 = fst(\hat{c}_1) \\ & & \vdots \\ & & \overline{c}'_k = fst(\hat{c}_k) \\ & & \overline{c} \leftarrow szFlat_k(\overline{c}'_1, \ldots, \overline{c}'_k, \overline{c}_1, \ldots, \overline{c}_k) \\ & & \overline{c}''_1 \leftarrow segsum(\overline{c}'_1, \overline{c}_1) \\ & & \vdots \\ & & \overline{c}''_k \leftarrow segsum(\overline{c}'_k, \overline{c}_k) \\ & & \hat{c}'_1 = snd(\hat{c}_1) \\ & & \vdots \\ & & \hat{c}'_k = snd(\hat{c}_k) \\ & & \hat{c} \leftarrow segzip_k(\hat{c}'_1, \ldots, \hat{c}'_k, \overline{c}''_1, \ldots, \overline{c}''_k) \\ & in & \mathsf{R}_1\ (\overline{c}, \hat{c}) \end{array}$$

**Work Complexity** If $\#\overline{c}_1 = \cdots = \#\overline{c}_k = l$ then the work complexity of $segzip_k(\hat{c}_1, \ldots, \hat{c}_k, \overline{c}_1, \ldots, \overline{c}_k)$ can be shown, by solving a recurrence, to be no greater than

$$5 \cdot \sum_{i=1}^{k} \|\hat{c}_i\| + K_1 \cdot \max_{i=1}^{k} \delta(\hat{c}_i) + K_2$$

$$K_1 = 1 + 3l + k + k^2 \qquad K_2 = l + k + k^2$$

**Normalization** Target language values with general segment descriptors can be normalized to values with normalized segment descriptors. Normalization is used for two purposes in this report. It is used to simplify the denotational semantics of the transformation of the list constructor, and it is used in reasoning about value preservation for vectorized programs in general. In the first use it is beneficial to have an associated work complexity with the normalization. In the second use it is not, since we do not want to charge for an extra normalization phase after each computation step. A value can be normalized including proper work complexity by the function *norm* and for free by the function $|\cdot|$.

$$norm : \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TValN}$$

$$norm([n_1, \ldots, n_l]) = \mathsf{R}_1\ [n_1, \ldots, n_l]$$

$$norm(\texttt{[]}) = \mathsf{R}_1\ \texttt{[]}$$

$$\begin{array}{lll} norm(((\overline{c}_1, \overline{c}_2), c)) = & let^{\$} & \hat{c} \leftarrow segfetch^{+}(c, \overline{c}_1, \overline{c}_2) \\ & in & \mathsf{R}_1\ (\overline{c}_2, \hat{c}) \end{array}$$

where $segfetch^{+}$ is a version of *segfetch* that will recurse in the nested case to fetch the entire value. Consequently the resulting value is normalized and the segment starts can be discarded.

$$segfetch^{+} : \mathbf{TVal} \times \mathbf{TVal} \times \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TValN}$$

$$segfetch^{+}(c, \overline{c}_1, \overline{c}_2) = \begin{cases} sfFlat(c, \overline{c}_1, \overline{c}_2) & (flat?c) \\ sfNested^{+}(c, \overline{c}_1, \overline{c}_2) & (pair?c) \\ \mathsf{R}_1\ \texttt{[]} & (empty?c\ and\ \overline{c}_2 = [0, \ldots, 0]) \end{cases}$$

$$sfNested^{+} : \mathbf{TVal} \times \mathbf{TVal} \times \mathbf{TVal} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TValN}$$

$$\begin{array}{lll} sfNested^{+}(((\overline{c}_1, \overline{c}_2), c), \overline{c}_3, \overline{c}_4) = & let^{\$} & \overline{c}'_1 \leftarrow sfFlat(\overline{c}_1, \overline{c}_3, \overline{c}_4) \\ & & \overline{c}'_2 \leftarrow sfFlat(\overline{c}_2, \overline{c}_3, \overline{c}_4) \\ & & \hat{c} \leftarrow segfetch^{+}(c, \overline{c}'_1, \overline{c}'_2) \\ & in & \mathsf{R}_1\ (\overline{c}'_2, \hat{c}) \end{array}$$

**Work Complexity** The work complexity of $norm(c)$ can be shown to be no greater than

$$1 + 4 \cdot \||c|\| + \delta(c)$$

$\||c|\|$ can be hard to read but it should be read as "the size of the normalization of $c$".

**Cost-free Normalization**

$$|\cdot| : \mathbf{TVal} \rightharpoonup \mathbf{TVal}$$

The $|\cdot|$ function is defined in the same way as the *norm* function but without monads. It will therefore be cost-free and will not be defined for erroneous input (type error or if a segment descriptor references out of bounds).

$$|{:}|\ \mathbf{TVal} \rightharpoonup \mathbf{TValN}$$

$$|[n_1,\ldots,n_l]| = [n_1,\ldots,n_l]$$
$$|\,\texttt{[]}\,| = \texttt{[]}$$
$$|((\bar{c}_1,\bar{c}_2),c)| = (\bar{c}_2, \triangle^{\bar{c}_1}_{\bar{c}_2}(c))$$

where $\triangle$ is the non-monadic version of *segfetch*$^+$.

$|\cdot|$ is also be defined for computations by the function $|\cdot|^\$$.

$$|\cdot|^\$ : \mathbf{M}^\$\ \mathbf{TVal} \rightharpoonup \mathbf{M}^\$\ \mathbf{TVal}$$

$$\left|c^\$\right|^\$ = \begin{array}{ll} let^\$ & c \leftarrow c^\$ \\ in & unit^\$\ \ |c| \end{array}$$

$\left|c^\$\right|^\$$ means the same as $c^\$$ but where the computed value is normalized for free. Furthermore $|\cdot|$ can be generalized to environments in the obvious way.

Finally normalization is also defined on types

$$|[\pi]| = [\pi]$$

$$|(\mathrm{seg},\sigma)| = ([\mathrm{int}], |\sigma|)$$

**Lemma 4** *Normalization type preservation.*

$$\vdash_{\mathcal{T}_\mathrm{s}} c : \sigma$$

$$if\ and\ only\ if\ \ \vdash_{\mathcal{T}_\mathrm{s}} |c| : |\sigma|$$

**Proof** By induction on $c$ and definition of $|\cdot|$.

**Lemma 5**

$$if\ \sum_{i=1}^{l} n_i = \#c$$

$$then\ \triangle^{[n_1,\ldots,n_l]}_{scan_+([n_1,\ldots,n_l])}(c) = |c|$$

**Proof** By induction on $c$ and definition of $\triangle$ and $|\cdot|$.

**De-normalization** The function *deNorm* takes a target language value where the segment descriptors only contain lengths and return a value where the segment descriptors contains starts and lengths by computing a plus-scan of the lengths.

$$deNorm : \mathbf{TValN} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$$deNorm(\hat{c}) = \begin{cases} \mathsf{R}_1\ \hat{c} & flat?(\hat{c}) \\ \mathsf{R}_1\ \texttt{[]} & empty?(\hat{c}) \\ & \\ let^{\$} \quad c \leftarrow makeStarts(\overline{c}) & \\ \qquad\quad c' \leftarrow deNorm(\hat{c}') & (\hat{c} = (\overline{c}_1, \hat{c}')) \\ in \quad\ \ \mathsf{R}_1\ (c, c') & \end{cases}$$

Work complexity: $1 + \|\hat{c}\| + \delta(\hat{c})$

**Expressions** The current parallel degree will be given as a parameter to the semantic function. It is invariantly greater than 0, since a parallel degree of 0 corresponds to no computation at all.

$$\mathbf{TParDeg} = \mathrm{N}^{+}$$

The denotational semantics of target language expressions is divided into four different functions - one for each syntactic category:

$$\mathcal{T}_{\mathrm{s}} : \mathbf{TExp}_{\mathrm{s}} \times \mathbf{TParDeg} \times (\mathbf{Var} \rightharpoonup_{\mathrm{fin}} \mathbf{TVal}) \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$$\mathcal{T}_{\mathrm{t}} : \mathbf{TExp}_{\mathrm{t}} \times (\mathbf{Var} \rightharpoonup_{\mathrm{fin}} \mathbf{TVal}) \rightharpoonup \mathbf{TVal}$$

$$\mathcal{U} : \mathbf{TScalarExp} \times (\mathbf{Var} \rightharpoonup_{\mathrm{fin}} \mathbf{TVal}) \rightharpoonup \mathrm{N}$$

$$\mathcal{P} : \mathbf{TOp} \times \mathbf{TParDeg} \times \mathbf{TExp}_{\mathrm{t}}^{*} \rightharpoonup \mathbf{M}^{\$}\ \mathbf{TVal}$$

$\phi$ will range over value environments $\mathbf{Var} \rightharpoonup_{\mathrm{fin}} \mathbf{TVal}$.

$\boxed{\mathcal{T}_{\mathrm{t}}\, [\![t]\!]\, \phi = c}$

$$\begin{aligned}
\mathcal{T}_{\mathrm{t}}\, [\![x]\!]\, \phi &= \phi(x) \\
\mathcal{T}_{\mathrm{t}}\, [\![\mathbf{attsd}(t_1, t_2)]\!]\, \phi &= (\mathcal{T}_{\mathrm{t}}\, [\![(t_1)]\!]\, \phi, \mathcal{T}_{\mathrm{t}}\, [\![t_2]\!]\, \phi) \\
\mathcal{T}_{\mathrm{t}}\, [\![\mathbf{data}(t)]\!]\, \phi &= snd(\mathcal{T}_{\mathrm{t}}\, [\![t]\!]\, \phi) \\
\mathcal{T}_{\mathrm{t}}\, [\![\mathbf{seg}(t)]\!]\, \phi &= fst(\mathcal{T}_{\mathrm{t}}\, [\![t]\!]\, \phi) \\
\mathcal{T}_{\mathrm{t}}\, [\![\mathbf{starts}(t)]\!]\, \phi &= fst(\mathcal{T}_{\mathrm{t}}\, [\![t]\!]\, \phi) \\
\mathcal{T}_{\mathrm{t}}\, [\![\mathbf{lens}(t)]\!]\, \phi &= snd(\mathcal{T}_{\mathrm{t}}\, [\![t]\!]\, \phi)
\end{aligned}$$

$\boxed{\mathcal{T}_{\mathrm{s}}\left[\!\left[s\right]\!\right]_l \phi = c^{\$}}$

$$
\begin{array}{lcl}
\mathcal{T}_{\mathrm{s}}\left[\!\left[t\right]\!\right]_l \phi & = & \mathsf{R}_1\ \mathcal{T}_{\mathrm{t}}\left[\!\left[t\right]\!\right] \phi \\[1em]
\mathcal{T}_{\mathrm{s}}\left[\!\left[[\overline{r}]\right]\!\right]_l \phi & = & \mathsf{R}_l\ l \bullet r \\[1em]
\mathcal{T}_{\mathrm{s}}\left[\!\left[\mathbf{let}\ x \Leftarrow s_1\ \mathbf{in}\ s_2\right]\!\right]_l \phi & = & \begin{array}[t]{ll} \mathit{let}^{\$} & c_1 \leftarrow \mathcal{T}_{\mathrm{s}}\left[\!\left[s_1\right]\!\right]_l \phi \\ \mathit{in} & \mathcal{T}_{\mathrm{s}}\left[\!\left[s_2\right]\!\right]_l \phi[x \mapsto c_1] \end{array} \\[1.5em]
\mathcal{T}_{\mathrm{s}}\left[\!\left[p(t_1,\ldots,t_k)\right]\!\right]_l \phi & = & \mathcal{P}\left[\!\left[p\right]\!\right]_l (\mathcal{T}_{\mathrm{t}}\left[\!\left[t_1\right]\!\right] \phi, \ldots, \mathcal{T}_{\mathrm{t}}\left[\!\left[t_k\right]\!\right] \phi) \\[1em]
\mathcal{T}_{\mathrm{s}}\left[\!\left[\mathbf{par}\ u\ \mathbf{do}\ s\right]\!\right]_l \phi & = & \begin{array}[t]{ll} \mathit{let}^{\$} & l' = \mathcal{U}\left[\!\left[u\right]\!\right] \phi \\ \mathit{in} & \begin{array}[t]{ll} \mathit{if} & l' = 0 \\ \mathit{then} & \mathsf{R}_1\ \texttt{[]} \\ \mathit{else} & \mathcal{T}_{\mathrm{s}}\left[\!\left[s\right]\!\right]_{l'} \phi \end{array} \end{array}
\end{array}
$$

$\boxed{\mathcal{U}\left[\!\left[u\right]\!\right] \phi = n}$

$$\mathcal{U}\left[\!\left[\mathbf{length}(x)\right]\!\right] \phi = \#\phi(x)$$

**Operations** $\boxed{\mathcal{P}\left[\!\left[p\right]\!\right]_l (c_1,\ldots,c_k) = c^{\$}}$

$$\mathcal{P}\left[\!\left[\mathbf{mkseg}\right]\!\right]_l (\overline{c}) = \mathit{makeStarts}(\overline{c})$$

$$\mathcal{P}\left[\!\left[\mathbf{ixrep}\right]\!\right]_l ([n_1,\ldots,n_l]) = \mathsf{R}_{l+\sum_{i=1}^{l} n_i}\ \mathit{ixrep}_l(n_1,\ldots,n_l)$$

$$\mathcal{P}\left[\!\left[\mathbf{iotas}\right]\!\right]_l ([n_1,\ldots,n_l]) = \mathsf{R}_{l+\sum_{i=1}^{l} n_i}\ \mathit{iotas}_l(n_1,\ldots,n_l)$$

$$\mathcal{P}\left[\!\left[\mathbf{segsum}\right]\!\right]_l (\overline{c}_1,\overline{c}_2) = \mathit{segsum}(\overline{c}_1,\overline{c}_2) \qquad (\#\overline{c}_1 = l)$$

$$\mathcal{P}\left[\!\left[\oplus^{\mathsf{v}}\right]\!\right]_l ([r_{1,1},\ldots,r_{1,l}],\ldots,[r_{k,1},\ldots,r_{k,l}]) = \mathsf{R}_l\left[\oplus(r_{1,1},\ldots,r_{k,1}),\ldots,\oplus(r_{1,l},\ldots,r_{k,l})\right]$$

$$
\mathcal{P}\left[\!\left[\mathbf{zip}_n\right]\!\right]_l (c_1,\ldots,c_k) = \begin{array}[t]{ll} \mathit{let}^{\$} & \hat{c}_1 \leftarrow \mathit{norm}(c_1) \\ & \vdots \\ & \hat{c}_k \leftarrow \mathit{norm}(c_k) \\ & \overline{c} \leftarrow \mathsf{R}_l\ [1]_l \\ & \hat{c} \leftarrow \mathit{segzip}_k(\hat{c}_1,\ldots,\hat{c}_k,\overline{c},\ldots,\overline{c}) \\ \mathit{in} & \mathit{deNorm}(\hat{c}) \end{array}
$$

$$\mathcal{P}\left[\!\left[\mathbf{segfetch}\right]\!\right]_l (c,\overline{c}_1,\overline{c}_2) = \mathit{segfetch}(c,\overline{c}_1,\overline{c}_2) \qquad (\#\overline{c}_1 = \#\overline{c}_2)$$

### 3.3.4 Properties

**Lemma 6** *Type Soundness.*

$$\textit{if}\ \Pi \vdash_{\mathcal{T}_s} s : \sigma\ \textit{and}\ \vdash_{\mathcal{T}} \phi : \Pi$$

$$\textit{then}\quad \textit{either}\quad \mathcal{T}_s [\![s]\!]_l\, \phi = \mathsf{E}$$

$$\textit{or}\quad \mathcal{T}_s [\![s]\!]_l\, \phi = \mathsf{R}_w\ c \quad \textit{and} \quad \vdash_{\mathcal{T}_s} c : \sigma$$

**Proof** By induction on $s$.

**Lemma 7** *Restriction to free variables.*

$$\mathcal{T}_s [\![s]\!]_l\, \phi = \mathcal{T}_s [\![s]\!]_l\, \phi \restriction FV(s)$$

**Proof** By induction on $s$.

## 3.4 Transformation

### 3.4.1 Types

$$\ll \cdot \gg : \mathbf{STyp} \rightarrow \mathbf{TTypGen} \qquad \ll \cdot \gg^{+} : \mathbf{STyp} \rightarrow \mathbf{TTypNorm}$$

$$\boxed{\ll \tau \gg = \dot{\sigma}}$$

$$\ll \pi \gg = [\pi]$$

$$\ll [\tau] \gg = (\text{seg}, \ll \tau \gg)$$

$$\boxed{\ll \tau \gg^{+} = \sigma^{+}}$$

$$\ll \pi \gg^{+} = [\pi]$$

$$\ll [\tau] \gg^{+} = ([\text{int}], \ll \tau \gg^{+})$$

$\ll \cdot \gg$ is extended to environments: $\boxed{\ll \Gamma \gg = \Pi}$

$$\ll [x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_k] \gg = [x_1 \mapsto \ll \tau_1 \gg, \ldots, x_k \mapsto \ll \tau_k \gg]$$

### 3.4.2 Values

The transformation of values (also called the representation function) is defined as a $l$-ary function. Source language parallel programs are evaluated separately in each parallel instance while target language parallel programs are evaluated uniformly across all parallel instances. As a consequence we will often be comparing multiple small source language values to one large target language variable. The arbitrary arity of the representation function allows us to do so elegantly.

$$\left\langle \cdot \right\rangle_l : \mathbf{SVal}^l \to \mathbf{TVal} \qquad (l \geq 0)$$

$$\boxed{\left\langle v_1, \ldots, v_l \right\rangle_l = c}$$

$$\left\langle \right\rangle_0 = \texttt{[]}$$

$$\left\langle n_1, \ldots, n_l \right\rangle_l = [n_1, \ldots, n_l] \qquad (l > 0)$$

$$\left\langle [v_1^1, \ldots, v_1^{l'_1}], \ldots, [v_l^1, \ldots, v_l^{l'_l}] \right\rangle_l \qquad (l > 0)$$

$$= \Big( [l_1, \ldots, l'_l]), \left\langle v_1^1, \ldots, v_1^{l'_1}, \ldots, v_l^1, \ldots, v_l^{l'_l} \right\rangle_{\sum_{i=1}^{l} n_i} \Big)$$

The representation function is also defined for source language computations where it will sum up the work of each argument.

$$\left\langle \cdot \right\rangle_l^{\$} : (\mathbf{M}^{\$}\ \mathbf{SVal})^l \to \mathbf{M}^{\$}\ \mathbf{TVal} \qquad (l \geq 0)$$

$$\boxed{\left\langle v_1^{\$}, \ldots, v_k^{\$} \right\rangle_k^{\$} = c^{\$}}$$

$$\begin{array}{rll} \left\langle c_1^{\$}, \ldots, c_k^{\$} \right\rangle_k^{\$} = & let^{\$} & c_1 \leftarrow c_1^{\$} \\ & & \vdots \\ & & c_k \leftarrow c_k^{\$} \\ & in & unit^{\$} \left\langle c_1, \ldots, c_k \right\rangle_k \end{array}$$

$\left\langle \cdot \right\rangle_l$ can be generalized to environments given that the same variables are mapped to values of the same type in each state:

$$\boxed{\left\langle \rho_1, \ldots, \rho_l \right\rangle_l = \phi}$$

$$\left\langle [x_1 \mapsto v_{1,1}, \ldots, x_k \mapsto v_{1,k}], \ldots, [x_1 \mapsto v_{l,1}, \ldots, x_k \mapsto v_{l,k}] \right\rangle_l$$

$$= [x_1 \mapsto \left\langle v_{1,1}, \ldots, v_{l,1} \right\rangle_l, \ldots, x_l \mapsto \left\langle v_{1,k}, \ldots, v_{l,k} \right\rangle_l]$$

**Lifting** Lifting in this context means the replication of data.

**Values**

$$lift_{[n_1, \ldots, n_k]}([r_1, \ldots, r_k]) = n_1 \bullet r_1 \mathbin{\char`\^} \cdots \mathbin{\char`\^} n_k \bullet r_k \qquad (\forall i. n_i \geq 0)$$

$$lift_{\overline{c}}(((\overline{c}_1, \overline{c}_2), c)) = ((lift_{\overline{c}}(\overline{c}_1), lift_{\overline{c}}(\overline{c}_2)), c)$$

**Finite Maps**

$$lift^X_{\overline{c}}(f)(x) = \begin{cases} lift_{\overline{c}}(f(x)) & x \in X \\ f(x) & x \notin X \end{cases}$$

**Expressions**

$$lift^{\{x_1,\ldots,x_k\}}_x(s) = \begin{array}{ll} \textbf{let} & \_x_1 = [1] \\ & \_x_2 = \textbf{ixrep}(x) \\ & x_1 \Leftarrow \textbf{segfetch}(x_1, \_x_2, \_x_1) \\ & \quad\vdots \\ & x_k \Leftarrow \textbf{segfetch}(x_k, \_x_2, \_x_1) \\ \textbf{in} & s \end{array}$$

**Lemma 8** *Lifting expression type preservation*

*If* $\Pi \vdash_{\mathcal{T}_s} s : \sigma$ *and* $\Pi(x) = [int]$ *and* $X \subseteq dom(\Pi)$ *and* $x \notin X$

*then* $\Pi \vdash_{\mathcal{T}_s} lift^X_x(s) : \sigma$

**Proof** By definition of $lift$ and $\vdash_{\mathcal{T}_s}$ and Lemma 3.

### 3.4.3 Expressions

$\langle\!\langle \cdot \rangle\!\rangle : \textbf{SExp} \rightarrow \textbf{TExp}_s$

$\boxed{\langle\!\langle e \rangle\!\rangle = s}$

$$\langle\!\langle \overline{r} \rangle\!\rangle = [\overline{r}]$$

$$\langle\!\langle x \rangle\!\rangle = x$$

$$\langle\!\langle \textbf{let } x \Leftarrow e_0 \textbf{ in } e_1 \rangle\!\rangle = \textbf{let } x \Leftarrow \langle\!\langle e_0 \rangle\!\rangle \textbf{ in } \langle\!\langle e_1 \rangle\!\rangle$$

$$\langle\!\langle o(e_1,\ldots,e_k) \rangle\!\rangle = \begin{array}{ll} \textbf{let} & \_x_1 \Leftarrow \langle\!\langle e_1 \rangle\!\rangle \\ & \quad\vdots \\ & \_x_k \Leftarrow \langle\!\langle e_k \rangle\!\rangle \\ \textbf{in} & \mathcal{O}\langle\!\langle o \rangle\!\rangle(\_x_1,\ldots,\_x_k) \end{array}$$

$$\langle\!\langle [e_0 : x \in \textbf{ix}(e_1)] \rangle\!\rangle = \begin{array}{ll} \textbf{let} & \_x_1 \Leftarrow \langle\!\langle e_1 \rangle\!\rangle \\ & x \Leftarrow \textbf{iotas}(\_x_1) \\ & \_x_2 \Leftarrow \textbf{mkseg}(\_x_1) \\ & \_x_3 \Leftarrow \textbf{par length}(x) \textbf{ do } lift^{FV(e_0)\setminus\{x\}}_{\_x_1}(\langle\!\langle e_0 \rangle\!\rangle) \\ \textbf{in} & \textbf{attsd}(\_x_2, \_x_3) \end{array}$$

**Operations**

$$\mathcal{O}\langle\!\langle \cdot \rangle\!\rangle : \mathbf{SOp} \times \mathbf{Var}^k \rightharpoonup \mathbf{TExp}_{\mathrm{s}}$$

$$\boxed{\mathcal{O}\langle\!\langle o \rangle\!\rangle(x_1, \ldots, x_k) = t}$$

$$\mathcal{O}\langle\!\langle \oplus \rangle\!\rangle(x_1, \ldots, x_k) = \oplus^{\mathrm{v}}(x_1, \ldots, x_k)$$

$$\mathcal{O}\langle\!\langle \mathbf{length} \rangle\!\rangle(x) = \mathbf{lens}(\mathbf{seg}(x))$$

$$\mathcal{O}\langle\!\langle \mathbf{elt} \rangle\!\rangle(x_1, x_2) = \begin{array}{ll} \mathbf{let} & \_x_1 \Leftarrow x_2 + \mathbf{starts}(\mathbf{seg}(x_1)) \\ & \_x_2 \Leftarrow [1] \\ \mathbf{in} & \mathbf{segfetch}(\mathbf{data}(x_1), \_x_1, \_x_2) \end{array}$$

$$\mathcal{O}\langle\!\langle \mathbf{vec}_k \rangle\!\rangle(x_1, \ldots, x_k) = \begin{array}{ll} \mathbf{let} & \_x_1 \Leftarrow [k] \\ & \_x_2 \Leftarrow \mathbf{mkseg}(\_x_1) \\ & \_x_3 \Leftarrow \mathbf{zip}_k(x_1, \ldots, x_k) \\ \mathbf{in} & \mathbf{attsd}(\_x_2, \_x_3) \end{array}$$

$$\mathcal{O}\langle\!\langle \mathbf{conc} \rangle\!\rangle(x) = \begin{array}{ll} \mathbf{let} & \_x_1 \Leftarrow \mathbf{segsum}(\mathbf{lens}(\mathbf{seg}(x)), \mathbf{lens}(\mathbf{seg}(\mathbf{data}(x)))) \\ & \_x_2 \Leftarrow \mathbf{mkseg}(\_x_1) \\ & \_x_3 \Leftarrow \mathbf{segfetch}(\mathbf{data}(\mathbf{data}(x)), \mathbf{starts}(\mathbf{seg}(\mathbf{data}(x))), \mathbf{lens}(\mathbf{seg}(\mathbf{data}(x)))) \\ \mathbf{in} & \mathbf{attsd}(\_x_2, \_x_3) \end{array}$$

$$\mathcal{O}\langle\!\langle \mathbf{part} \rangle\!\rangle(x_1, x_2) = \begin{array}{ll} \mathbf{let} & \_x_1 \Leftarrow \mathbf{mkseg}(\mathbf{data}(x_2)) \\ & \_x_2 \Leftarrow \mathbf{attsd}(\_x_1, \mathbf{data}(x_1)) \\ \mathbf{in} & \mathbf{attsd}(\mathbf{seg}(x_2), \_x_2) \end{array}$$

## 3.5 Transformation Properties

**Lemma 9** *Operation Type Preservation*

$$\forall o. \forall \tau'_1..\tau'_{k'}.$$

$$\begin{array}{ll} \mathit{if} & o[\tau'_1, \ldots, \tau'_{k'}] : \tau_1 * \cdots * \tau_k \; {-}{>} \; \tau \\ & \mathcal{T} \\ \mathit{then} & \forall\, \Pi. \quad \Pi[\_x_1 \mapsto \ll \tau_1 \gg, \ldots, \_x_k \mapsto \ll \tau_k \gg] \vdash_{\mathcal{T}_{\mathrm{s}}} \mathcal{O}\langle\!\langle o \rangle\!\rangle(\_x_1, \ldots, \_x_k) : \; \ll \tau \gg \end{array}$$

**Proof** By cases of $o$

- Case $o = \oplus$:

  We have $\oplus : \pi_1 * \cdots * \pi_k \; {-}{>} \; \pi$

  $\ll \pi \gg = [\pi]$

  $\ll \pi_1 \gg = [\pi_1] \quad \cdots \quad \ll \pi_k \gg = [\pi_k]$

  $\mathcal{O}\langle\!\langle \oplus \rangle\!\rangle(\_x_1, \ldots, \_x_k) = \oplus^{\mathrm{v}}(\_x_1, \ldots, \_x_k)$

  $\oplus^{\mathrm{v}} : [\pi_1] * \cdots * [\pi_k] \; {-}{>} \; [\pi]$

Let $\Pi' = \Pi[\_x_1 \mapsto [\pi_1] .. \_x_k \mapsto [\pi_k]]$. Then by (TOP-T)

$$\mathcal{T} = \frac{\overline{\Pi' \vdash_{\mathcal{T}_s} \_x_1 : [\pi_1]} \quad \cdots \quad \overline{\Pi' \vdash_{\mathcal{T}_s} \_x_k : [\pi_k]}}{\Pi' \vdash_{\mathcal{T}_s} \oplus^{\mathrm{v}}(\_x_1, \ldots, \_x_k) : [\pi]} (\oplus^{\mathrm{v}} : [\pi_1] * \cdots * [\pi_k] \mathrel{->} [\pi])$$

as required.

- Case $o = \mathbf{length}$:

  We have $\mathbf{length}[\tau'] : [\tau'] \mathrel{->} \mathrm{int}$, so $k = 1$, $\tau_1 = [\tau']$ and $\tau = \mathrm{int}$.

  $\ll \mathrm{int} \gg = [\mathrm{int}]$

  $\ll [\tau'] \gg = (\mathrm{seg}, \ll \tau \gg')$

  $\langle\!\langle \mathbf{length} \rangle\!\rangle(\_x_1) = \mathbf{lens}(\mathbf{seg}(\_x_1))$

  By (TSEG-T) and (TSTARTS-T) we have

$$\mathcal{T} = \frac{\dfrac{\overline{\Pi[\_x \mapsto (\mathrm{seg}, \ll \tau \gg')] \vdash_{\mathcal{T}_s} \_x_1 : (\mathrm{seg}, \ll \tau \gg')}}{\Pi[\_x \mapsto (\mathrm{seg}, \ll \tau \gg')] \vdash_{\mathcal{T}_s} \mathbf{seg}(\_x_1) : \mathrm{seg}}}{\Pi[\_x \mapsto (\mathrm{seg}, \ll \tau \gg')] \vdash_{\mathcal{T}_s} \mathbf{starts}(\mathbf{seg}(\_x_1)) : [\mathrm{int}]}$$

- Case $o = \mathbf{vec}_k$:

  We have $\mathbf{vec}_k[\tau'] : {\tau'}^k \mathrel{->} [\tau']$

  $\ll [\tau'] \gg = (\mathrm{seg}, \ll \tau' \gg)$

$$\begin{array}{rll} \mathcal{O}\langle\!\langle \mathbf{vec}_k \rangle\!\rangle(\_x_1, \ldots, \_x_k) = & \mathbf{let} & \_x'_1 \Leftarrow [\overline{k}] \\ & & \_x'_2 \Leftarrow \mathbf{mkseg}(\_x'_1) \\ & & \_x'_3 \Leftarrow \mathbf{zip}_k(\_x_1, \ldots, \_x_k) \\ & \mathbf{in} & \mathbf{attsd}(\_x'_2, \_x'_3) \end{array}$$

  Let $\Pi' = \Pi[\_x_1 \mapsto \ll \tau' \gg .. \_x_k \mapsto \ll \tau' \gg]$. By (TLET-T) and (TNUMV-T) we have.

$$\mathcal{T} = \frac{\overline{\Pi' \vdash_{\mathcal{T}_s} [\overline{k}] : [\mathrm{int}]} \qquad \dfrac{\mathcal{T}'}{\Pi'[\_x'_1 \mapsto [\mathrm{int}]] \vdash_{\mathcal{T}_s} s' : (\mathrm{seg}, \ll \tau' \gg)}}{\Pi' \vdash_{\mathcal{T}_s} \mathbf{let}\ \_x'_1 \Leftarrow [\overline{k}]\ \mathbf{in}\ s' : (\mathrm{seg}, \ll \tau' \gg)}$$

  Let $\Pi'' = \Pi'[\_x'_1 \mapsto [\mathrm{int}]]$. By (TLET-T), (TVAR-T) and (TOP-T) with $\mathbf{mkseg} : [\mathrm{int}] \mathrel{->} \mathrm{seg}$ we have

$$\mathcal{T}' = \frac{\dfrac{\overline{\Pi'' \vdash_{\mathcal{T}_s} \_x'_1 : [\mathrm{int}]}}{\Pi'' \vdash_{\mathcal{T}_s} \mathbf{mkseg}(\_x'_1) : \mathrm{seg}} \qquad \dfrac{\mathcal{T}''}{\Pi''[\_x'_2 \mapsto \mathrm{seg}] \vdash_{\mathcal{T}_s} s'' : (\mathrm{seg}, \ll \tau' \gg)}}{\Pi'' \vdash_{\mathcal{T}_s} \mathbf{let}\ \_x'_2 \Leftarrow \mathbf{mkseg}(\_x'_1)\ \mathbf{in}\ s'' : (\mathrm{seg}, \ll \tau' \gg)}$$

Let $\Pi''' = \Pi''[\_x'_2 \mapsto \text{seg}]$. By (TVar-T), (TOp-T) with $\mathbf{zip}_k[\ll\tau'\gg] : \ll\tau'\gg^k \;\texttt{->}\; \ll\tau'\gg$ and (TLet-T) we have

$$\mathcal{T}'' = \cfrac{\cfrac{\cfrac{}{\Pi''' \vdash_{\mathcal{T}_s} \_x_1 : \ll\tau'\gg} \quad \cdots \quad \cfrac{}{\Pi''' \vdash_{\mathcal{T}_s} \_x_k : \ll\tau'\gg}}{\Pi''' \vdash_{\mathcal{T}_s} \mathbf{zip}_k(\_x_1, \ldots, \_x_k) : \ll\tau'\gg} \qquad \begin{array}{c}\mathcal{T}''' \\ \Pi'''[\_x'_3 \mapsto \ll\tau'\gg] \vdash_{\mathcal{T}_s} s''' : (\text{seg}, \ll\tau'\gg)\end{array}}{\Pi''' \vdash_{\mathcal{T}_s} \mathbf{let}\ \_x'_2 \Leftarrow \mathbf{zip}_k(\_x_1, \ldots, \_x_k)\ \mathbf{in}\ s'' : (\text{seg}, \ll\tau'\gg)}$$

Let $\Pi'''' = \Pi'''[\_x'_3 \mapsto \ll\tau'\gg]$

$$\mathcal{T}''' = \cfrac{\cfrac{}{\Pi'''' \vdash_{\mathcal{T}_s} \_x'_2 : \text{seg}} \quad \cfrac{}{\Pi'''' \vdash_{\mathcal{T}_s} \_x'_3 : \ll\tau'\gg}}{\Pi'''' \vdash_{\mathcal{T}_s} \mathbf{attsd}(\_x'_2, \_x'_3) : (\text{seg}, \ll\tau'\gg)}$$

- Cases $o = \mathbf{elt}$, $o = \mathbf{conc}$ and $o = \mathbf{part}$ are similar.

□

**Lemma 10** *Expression Type Preservation.*

$\forall e.\forall\Gamma.$

*if* $\quad \Gamma \vdash_{\mathcal{S}} e : \tau$

*then* $\quad \ll\Gamma\gg \vdash_{\mathcal{T}_s} \langle\!\langle e \rangle\!\rangle : \ll\tau\gg$

**Proof** By induction on the syntax of $e$:

- Case $e = \overline{r}$: By definition $\langle\!\langle e \rangle\!\rangle = [\overline{r}]$, and by (TNumV-T) we have

  $$\cfrac{}{\Pi \vdash_{\mathcal{T}_s} [\overline{r}] : [\text{int}]}$$

  , so in particular $\ll\Gamma\gg \vdash_{\mathcal{T}_s} \langle\!\langle e \rangle\!\rangle : [\text{int}]$.

  By (SNum-T) we have that $\Gamma \vdash_{\mathcal{S}} \overline{r} : \text{int}$, so $\tau = \text{int}$. We also have $\ll\text{int}\gg = [\text{int}]$ as required.

- Case $e = x$: By definition of $\langle\!\langle \cdot \rangle\!\rangle$ we have that $\langle\!\langle e \rangle\!\rangle = x$, and by (TVar-T) we have that $\ll\Gamma\gg \vdash_{\mathcal{T}_s} x : \ll\Gamma\gg(x)$.

  By definition of $\ll\cdot\gg$ we have $\ll\Gamma\gg(x) = \ll\Gamma(x)\gg$, so $\ll\Gamma\gg \vdash_{\mathcal{T}_s} \langle\!\langle e \rangle\!\rangle : \ll\Gamma(x)\gg$.

  By (SVar-T) we have that $\Gamma \vdash_{\mathcal{S}} e : \Gamma(x)$, so $\tau = \Gamma(x)$ and $\ll\Gamma\gg \vdash_{\mathcal{T}_s} \langle\!\langle e \rangle\!\rangle : \ll\tau\gg$.

- Case $e = \mathbf{let}\ x \Leftarrow e_0\ \mathbf{in}\ e_1$:

  By (SLet-T) we must have a derivation of

  $$\cfrac{\begin{array}{c}\mathcal{S}_0 \\ \Gamma \vdash_{\mathcal{S}} e_0 : \tau_0\end{array} \qquad \begin{array}{c}\mathcal{S}_1 \\ \Gamma[x \mapsto \tau_0] \vdash_{\mathcal{S}} e_1 : \tau\end{array}}{\Gamma \vdash_{\mathcal{S}} \mathbf{let}\ x \Leftarrow e_0\ \mathbf{in}\ e_1 : \tau}$$

By I.H on $\mathcal{S}_0$ we have that

$$\ll\Gamma\gg\vdash_{\mathcal{T}_s} \overset{\mathcal{T}_0}{\langle\!\langle e_0 \rangle\!\rangle : \ll\tau_0\gg}$$

By I.H on $\mathcal{S}_1$ we have that $\ll\Gamma[x \mapsto \tau_0]\gg\vdash_{\mathcal{T}_s} \langle\!\langle e_1 \rangle\!\rangle : \ll\tau\gg$, and by definition of $\ll\cdot\gg$ we have that $\ll\Gamma[x \mapsto \tau_0]\gg=\ll\Gamma\gg [x \mapsto\ll\tau_0\gg]$, so

$$\overset{\mathcal{T}_1}{\ll\Gamma\gg [x \mapsto\ll\tau_0\gg] \vdash_{\mathcal{T}_s} \langle\!\langle e_1 \rangle\!\rangle : \ll\tau\gg}$$

By (TLet-T) we have

$$\frac{\overset{\mathcal{T}_0}{\ll\Gamma\gg \vdash_{\mathcal{T}_s} \langle\!\langle e_0 \rangle\!\rangle : \ll\tau_0\gg} \qquad \overset{\mathcal{T}_1}{\ll\Gamma\gg[x \mapsto\ll\tau_0\gg] \vdash_{\mathcal{T}_s} \langle\!\langle e_1 \rangle\!\rangle : \ll\tau\gg}}{\ll\Gamma\gg \vdash_{\mathcal{T}_s} \mathbf{let}\ x \Leftarrow \langle\!\langle e_0 \rangle\!\rangle\ \mathbf{in}\ \langle\!\langle e_1 \rangle\!\rangle : \ll\tau\gg}$$

By definition of $\langle\!\langle \cdot \rangle\!\rangle$ we have that $\langle\!\langle e \rangle\!\rangle = \mathbf{let}\ x \Leftarrow \langle\!\langle e_0 \rangle\!\rangle\ \mathbf{in}\ \langle\!\langle e_1 \rangle\!\rangle$, so $\ll\Gamma\gg\vdash_{\mathcal{T}_s} \langle\!\langle e \rangle\!\rangle : \ll\tau\gg$.

- Case $e = o(e_1, \ldots, e_k)$:

  By (SOp-T) we have

$$\frac{\overset{\mathcal{S}_1}{\Gamma \vdash_{\mathcal{S}} e_1 : \tau_1} \quad \cdots \quad \overset{\mathcal{S}_k}{\Gamma \vdash_{\mathcal{S}} e_k : \tau_k}}{\Gamma \vdash_{\mathcal{S}} o(e_1, \ldots, e_k) : \tau} \,(o[\tau_1', \ldots, \tau_k'] : \tau_1 * \cdots * \tau_k \tau)$$

  We have

$$\begin{array}{rll} \langle\!\langle o(e_1, \ldots, e_k) \rangle\!\rangle = & \mathbf{let} & \_x_1 \Leftarrow \langle\!\langle e_1 \rangle\!\rangle \\ & & \vdots \\ & & \_x_k \Leftarrow \langle\!\langle e_k \rangle\!\rangle \\ & \mathbf{in} & \mathcal{O}\langle\!\langle o \rangle\!\rangle(\_x_1, \ldots, \_x_k) \end{array}$$

  Let $\Gamma' = \Gamma[\_x_1 \mapsto \tau_1, \ldots, \_x_k \mapsto \tau_k]$.

  Since $\_x_i$ is chosen s.t. the variable does not conflict with any variables in $\Gamma$ we have $\Gamma \subseteq \Gamma'$ and from lemma 1 on $\mathcal{S}_1 \cdots \mathcal{S}_k$ we must have the derivations:

$$\overset{\mathcal{S}_1'}{\Gamma' \vdash_{\mathcal{S}} e_1 : \tau_1} \quad \cdots \quad \overset{\mathcal{S}_k'}{\Gamma' \vdash_{\mathcal{S}} e_k : \tau_k}$$

  By IH on $\mathcal{S}_i'$ for $i \in \{1..k\}$ we have

$$\ll\Gamma'\gg\vdash_{\mathcal{T}_s} \overset{\mathcal{T}_i}{\langle\!\langle e_i \rangle\!\rangle : \ll\tau_i\gg}$$

By Lemma 9 (operation type preservation) we have

$$\ll\Gamma'\gg\vdash_{\mathcal{T}_s} \overset{\mathcal{T}'}{\mathcal{O}\langle\!\langle o\rangle\!\rangle(\_x_1,\ldots,\_x_k) :} \ll\tau\gg$$

By repeated use of (TLet-T) and lemma 3 we get

$$\frac{\overset{\mathcal{T}_k}{\ll\Gamma'\gg\vdash_{\mathcal{T}_s} \langle\!\langle e_k\rangle\!\rangle : \ll\tau_k\gg} \qquad \overset{\mathcal{T}'}{\ll\Gamma'\gg\vdash_{\mathcal{T}_s} \mathcal{O}\langle\!\langle o\rangle\!\rangle(\_x_1,\ldots,\_x_k) : \ll\tau\gg}}{\ll\Gamma'\gg\vdash_{\mathcal{T}_s} \textbf{let } x_k \Leftarrow \langle\!\langle e_k\rangle\!\rangle \textbf{ in } \mathcal{O}\langle\!\langle o\rangle\!\rangle(\_x_1,\ldots,\_x_k) : \ll\tau\gg}$$

- Case $e = [e_0 : x \in \textbf{ix}(e_1)]$:

  By (SApp-T) we must have a derivation of

$$\frac{\overset{\mathcal{S}_1}{\Gamma\vdash_{\mathcal{S}} e_1 : \text{int}} \qquad \overset{\mathcal{S}_0}{\Gamma[x\mapsto \text{int}]\vdash_{\mathcal{S}} e_0 : \tau'}}{\Gamma\vdash_{\mathcal{S}} [e_0 : x \in \textbf{ix}(e_1)] : [\tau']}$$

  so $\tau = [\tau']$ and $\ll\tau\gg = (\text{seg}, \ll\tau'\gg)$.

  By IH on $\mathcal{S}_1$ we have

$$\overset{\mathcal{T}_1}{\ll\Gamma\gg\vdash_{\mathcal{T}_s} e_1 : [\text{int}]}$$

  By IH on $\mathcal{S}_0$ we have

$$\overset{\mathcal{T}_0}{\ll\Gamma[x\mapsto \text{int}]\gg\vdash_{\mathcal{T}_s} e_0 : \ll\tau'\gg}$$

  The vectorized expression is

$$\langle\!\langle [e_0 : x \in \textbf{ix}(e_1)]\rangle\!\rangle = \begin{array}{ll} \textbf{let} & \_x_1 \Leftarrow \langle\!\langle e_1\rangle\!\rangle \\ & x \Leftarrow \textbf{iotas}(\_x_1) \\ & \_x_2 \Leftarrow \textbf{mkseg}(\_x_1) \\ & \_x_3 \Leftarrow \textbf{par length}(x) \textbf{ do } lift_{\_x_1}^{FV(e_0)\backslash\{x\}}(\langle\!\langle e_0\rangle\!\rangle) \\ \textbf{in} & \textbf{attsd}(\_x_2, \_x_3) \end{array}$$

  For the let-binded expressions we have the types

$$\overset{\mathcal{T}_1}{\ll\Gamma\gg\vdash_{\mathcal{T}_s} e_1 : [\text{int}]}$$

$$\frac{\overline{\ll\Gamma\gg [\_x_1\mapsto[\text{int}]]\vdash_{\mathcal{S}} \_x_1 : [\text{int}]}}{\ll\Gamma\gg [\_x_1\mapsto[\text{int}]]\vdash_{\mathcal{S}} \textbf{iotas}(\_x_1) : [\text{int}]} \; (\textbf{iotas} : [\text{int}] \mathrel{-\!\!>} [\text{int}])$$

$$\frac{\overline{\ll\Gamma\gg [\_x_1\mapsto[\text{int}], x\mapsto[\text{int}]]\vdash_{\mathcal{S}} \_x_1 : [\text{int}]}}{\ll\Gamma\gg [\_x_1\mapsto[\text{int}], x\mapsto[\text{int}]]\vdash_{\mathcal{S}} \textbf{mkseg}(\_x_1) : \text{seg}} \; (\textbf{mkseg} : [\text{int}]\mathrel{-\!\!>}\text{seg})$$

Let $\Pi = \ll\Gamma\gg [\_x_1 \mapsto [\text{int}], x \mapsto [\text{int}], \_x_2 \mapsto \text{seg}]$

By lemma 3 on $\mathcal{T}_0$ and by lemma 8 with $\Pi$ we have

$$\overset{\mathcal{T}_0'}{\Pi \vdash_{\mathcal{T}_s} lift_{\_x_1}^{FV(e_0)\backslash\{x\}}(\langle\!\langle e_0 \rangle\!\rangle) : \ \ll\tau'\gg}$$

So

$$\frac{\dfrac{\overline{\Pi \vdash_{\mathcal{T}_s} x : [\text{int}]}}{\Pi \vdash_{\mathcal{U}} \textbf{length}(x)} \qquad \overset{\mathcal{T}_0'}{\Pi \vdash_{\mathcal{T}_s} lift_{\_x_1}^{FV(e_0)\backslash\{x\}}(\langle\!\langle e_0 \rangle\!\rangle) : \ \ll\tau'\gg}}{\Pi \vdash_{\mathcal{T}_s} \textbf{par length}(x) \textbf{ do } lift_{\_x_1}^{FV(e_0)\backslash\{x\}}(\langle\!\langle e_0 \rangle\!\rangle) : \ \ll\tau'\gg}$$

$$\frac{\overline{\Pi[\_x_3 \mapsto \ll\tau'\gg] \vdash_{\mathcal{T}_s} \_x_2 : \text{seg}} \ \cdots \ \overline{\Pi[\_x_3 \mapsto \ll\tau'\gg] \vdash_{\mathcal{T}_s} \_x_3 : \ \ll\tau'\gg}}{\Pi[\_x_3 \mapsto \ll\tau'\gg] \vdash_{\mathcal{T}_s} \textbf{attsd}(\_x_2, \_x_3) : (\text{seg}, \ll\tau'\gg)}$$

By repeated use of the (TLet-T) rule we finally get

$$\ll\Gamma\gg \vdash_{\mathcal{T}_s} \langle\!\langle e \rangle\!\rangle : (\text{seg}, \ll\tau'\gg)$$

as required.

□

**Lemma 11** *Value Type Preservation.*

$$\textit{if} \qquad \overset{\mathcal{S}_1}{\vdash_{\mathcal{S}} v_1 : \tau} \quad \cdots \textit{ and } \cdots \quad \overset{\mathcal{S}_l}{\vdash_{\mathcal{S}} v_l : \tau} \qquad (l \geq 0)$$

$$\textit{then} \qquad \overset{\mathcal{T}}{\vdash_{\mathcal{T}_s} \langle v_1, \ldots, v_l \rangle_l : \ \ll\tau\gg^+}$$

**Proof** By induction on $\tau$:

- Case $\tau = \pi$:

  Sub-case $\pi = \text{int}$

  We must have

  $$\mathcal{S}_i = \frac{}{\vdash_{\mathcal{S}} \overline{n_i} : \text{int}}$$

  so $v_i = \overline{n_i}$.

  We have

  $$\ll\text{int}\gg = [\text{int}]$$

– Sub-sub-case $l > 0$:

$$\langle n_1, \ldots, n_l \rangle_l = [n_1, \ldots, n_l]$$

By (TNumV-T):

$$\frac{}{\vdash_{\mathcal{T}_s} [n_1, \ldots, n_l] : [\text{int}]}$$

– Sub-sub-case $l = 0$:

$$\langle n_1, \ldots, n_l \rangle_l = \texttt{[]}$$

By (Empty-T):

$$\frac{}{\vdash_{\mathcal{T}_s} \texttt{[]} : [\text{int}]}$$

Case $\tau = [\tau']$: By (SVec-T) we have

$$\mathcal{S}_i = \cfrac{\begin{array}{ccc} \mathcal{S}_{i,1} & & \mathcal{S}_{i,l'_i} \\ \vdash_{\mathcal{S}} v_{i,1} : \tau' & \cdots & \vdash_{\mathcal{S}} v_{i,l'_i} : \tau' \end{array}}{\vdash_{\mathcal{S}} [v_{i,1}, \ldots, v_{i,l'_i}] : [\tau']}$$

so $v_i = [v_{i,1}, \ldots, v_{i,l'_i}]$

We have

$$\ll [\tau'] \gg^+ = ([\text{int}], \ll \tau' \gg^+)$$

Let $l' = \sum_{i=1}^{l} l'_i$.

- – Sub-case $l' = 0$

$$\langle \overbrace{[], \ldots, []}^{l} \rangle = ([\overbrace{0, \ldots, 0}^{l}], \langle\rangle_0)$$

$$\frac{\frac{}{\vdash_{\mathcal{T}_s} [0, \ldots, 0] : [\text{int}]} \qquad \frac{}{\vdash_{\mathcal{T}_s} \texttt{[]} : \ll \tau \gg^+}}{\vdash_{\mathcal{T}_s} (([0, \ldots, 0], \texttt{[]}) : ([\text{int}], \ll \tau \gg^+)}$$

as required.

– Sub-case $l' > 0$

We have

$$\begin{array}{rcl} \langle v_1, \ldots, v_l \rangle & = & \langle [v_{1,1}, \ldots, v_{1,l_1}], \ldots, [v_{l,1}, \ldots, v_{l,l_l}] \rangle_l \\ & = & ([l_1, \ldots, l_l], \langle v_{1,1}, \ldots, v_{1,l_1}, \ldots, v_{l,1}, \ldots, v_{l,l_l} \rangle_{l'}) \end{array}$$

By IH on $\mathcal{S}_{i,j}$ we have

$$\overset{\mathcal{T}'}{\vdash_{\mathcal{T}_s} \left\langle v_{1,1}, \ldots, v_{l,l'_l} \right\rangle_{l'} : \ll\tau'\gg^+}$$

$$\frac{\overline{\vdash_{\mathcal{T}_s} [l_1, \ldots, l_i] : [\text{int}]} \qquad \overset{\mathcal{T}'}{\vdash_{\mathcal{T}_s} \left\langle v_{1,1}, \ldots, v_{1,l_1}, \ldots, v_{l,1}, \ldots, v_{l,l_l} \right\rangle_{l'} : \ll\tau\gg^+}}{\vdash_{\mathcal{T}_s} \left([l_1, \ldots, l_l], \left\langle v_{1,1}, \ldots, v_{1,l_1}, \ldots, v_{l,1}, \ldots, v_{l,l_l} \right\rangle_{l'}\right) : ([\text{int}], \ll\tau\gg^+)}$$

as required.

□

### 3.5.1 Congruence

For $K \in \mathrm{N}$ define a relation $(\lesssim_K) \subseteq \mathbf{M}^\$ \ \mathbf{TVal} \times \mathbf{M}^\$ \ \mathbf{TVal}$ by

$$\begin{array}{l} c_1^\$ \lesssim_K c_2^\$ \quad \textit{if and only if} \\ \quad c_1^\$ = \mathsf{E} \quad \textit{and} \quad c_2^\$ = \mathsf{E} \\ \qquad \textit{or} \\ \quad c_1^\$ = \mathsf{R}_{w_1}\ c_1 \quad \textit{and} \quad c_2^\$ = \mathsf{R}_{w_2}\ c_2 \\ \quad \textit{and}\ c_1 = c_2 \quad \textit{and} \quad w_1 \leq K \cdot w_2 \end{array}$$

When $K$ equals 1 the subscript will not be written.

**Lemma 12** *Lifting value and work preservation.*

$$\textit{if}\ \phi(x) = \textit{ixrep}_l(n_1, \ldots, n_l) \quad \textit{and} \quad \forall i.n_i \geq 0 \quad \textit{and} \quad l' = \sum_{i=1}^{l} n_i > 0$$

$$\begin{array}{l} \textit{then}\ \mathcal{T}_\mathrm{s} \left[\!\left[ \textit{lift}_x^X(t) \right]\!\right]_l \phi \\ \qquad \lesssim \quad \begin{array}{ll} \textit{let}^\$ & \_ \leftarrow \texttt{Tick}_{l+2l'+|X|(1+4l+4l')} \\ \textit{in} & \mathcal{T}_\mathrm{s} [\![t]\!]_{l'}\ \textit{lift}^X_{[n_1,\ldots,n_l]}(\phi) \end{array} \end{array}$$

**Proof** By definition of $\mathcal{T}_\mathrm{s}[\![\cdot]\!]$, $\textit{lift}$ and *segfetch*.

**Lemma 13** *Operation work and value preservation.*

$$\begin{array}{l} \textit{if}\ \ |c_1| = \left\langle v_1^1, \ldots, v_1^l \right\rangle_l \quad \cdots \quad |c_k| = \left\langle v_k^1, \ldots, v_k^l \right\rangle_l \\[1em] \textit{and} \quad \overset{\mathcal{S}_{1,1}}{\vdash_\mathcal{S} v_1^1 : \tau_1} \quad \cdots \quad \overset{\mathcal{S}_{l,k}}{\vdash_\mathcal{S} v_k^l : \tau_k} \\ \textit{and}\ o[\tau'_1, \ldots, \tau'_{k'}] : \tau_1 * \cdots * \tau_k \ {-}{>}\ \tau \\ \textit{then}\ \forall\phi. \left| \mathcal{T}_\mathrm{s} \left[\!\left[ \mathcal{O}\langle\!\langle o \rangle\!\rangle(\_x_1, \ldots, \_x_k) \right]\!\right]_l \phi[\_x_1 \mapsto c_1, \ldots, \_x_k \mapsto c_k] \right|^\$ \\ \qquad \lesssim_K \left\langle \mathcal{O}[\![o]\!]\ (v_1^1, \ldots, v_k^1), \ldots, \mathcal{O}[\![o]\!]\ (v_1^l, \ldots, v_k^l) \right\rangle_l^\$ \end{array}$$

**Proof** By lemma 2 (type soundness) we have an evaluation of

$$\mathcal{O}\,[\![o]\!]\,(v_1^1,\ldots,v_k^1) \qquad \cdots \qquad \mathcal{O}\,[\![o]\!]\,(v_1^l,\ldots,v_k^l)$$

By lemma 11 on $\mathcal{S}_{1,1}\cdots\mathcal{S}_{l,k}$ we have

$$\vdash_{\mathcal{T}_s} \langle v_1^1,\ldots,v_1^l\rangle_l : \ll\tau_1\gg^+ \quad \cdots \quad \vdash_{\mathcal{T}_s} \langle v_k^1,\ldots,v_k^l\rangle_l : \ll\tau_k\gg^+$$

and also

$$\vdash_{\mathcal{T}_s} |c_1| : \ll\tau_1\gg^+ \quad \cdots \quad \vdash_{\mathcal{T}_s} |c_k| : \ll\tau_k\gg^+$$

Therefore by lemma 4 we have

$$\vdash_{\mathcal{T}_s} c_1 : \ll\tau_1\gg \quad \cdots \quad \vdash_{\mathcal{T}_s} c_k : \ll\tau_k\gg$$

By lemma 9 on $o[\tau'_1,\ldots,\tau'_{k'}] : \tau_1 * \cdots * \tau_k \;\text{->}\; \tau$ we have

$$\Pi[\_x_1 \mapsto \ll\tau_1\gg,\ldots,\_x_k \mapsto \ll\tau_k\gg] \vdash_{\mathcal{T}_s} \mathcal{O}\langle\!\langle o\rangle\!\rangle(\_x_1,\ldots,\_x_k) : \ll\tau\gg$$

Therefore by $\vdash_{\mathcal{T}_s} c_1 : \ll\tau_1\gg \quad \cdots \quad \vdash_{\mathcal{T}_s} c_k : \ll\tau_k\gg$ and lemma 6 (type soundness) we have an evaluation of

$$\mathcal{T}_s\,[\![\mathcal{O}\langle\!\langle o\rangle\!\rangle(\_x_1,\ldots,\_x_k)]\!]_l\,\phi[\_x_1 \mapsto c_1,\ldots,\_x_k \mapsto c_k]$$

The rest of the proof proceeds by cases of $o$.

- Case $o = \oplus$:

  We have the type $\oplus : \pi_1 * \ldots * \pi_k \;\text{->}\; \pi$

  So we must have a derivation of

  $$\mathcal{S}_{i,j} = \frac{}{\vdash_{\mathcal{S}} r_j^i : \pi_j}$$

  so $v_j^i = r_j^i$ and $\tau_j = \pi_j$ for all $i \in \{1..l\}$ and $j \in \{1..k\}$.

  We therefore have

  $$|c_j| = \langle v_j^1,\ldots,v_j^l\rangle_l = [r_j^1,\ldots,r_k^l]$$

  so $c_j = [r_j^1,\ldots,r_k^l]$

  We have the evaluations

  $$\mathcal{O}\,[\![\oplus]\!]\,(r_1^i,\ldots,r_k^i) = \mathsf{R}_1\;\oplus(r_1^i,\ldots,r_k^i)$$

  We have the vectorization

  $$\mathcal{O}\langle\!\langle\,\oplus\,\rangle\!\rangle(\_x_1,\ldots,\_x_k) = \oplus^{\mathrm{v}}(\_x_1,\ldots,\_x_k)$$

  We have the vectorized evaluation

  $$\begin{aligned}&\mathcal{T}_s\,[\![\oplus^{\mathrm{v}}(\_x_1,\ldots,\_x_k)]\!]_l\,\phi[\_x_1 \mapsto c_1,\ldots,\_x_k \mapsto c_k]\\ &= \mathsf{R}_l\,\Big[\oplus(r_1^1,\ldots,r_k^1),\ldots,\oplus(r_1^l,\ldots,r_k^l)\Big]\end{aligned}$$

  Clearly we have congruence for any $K \geq 1$.

- Case $o = \textbf{length}$:
  We have the type

  $$\textbf{length}[\tau'] : [\tau'] \texttt{ -> } \mathrm{int}$$

  so $k = 1$ and $\tau_1 = [\tau']$ From $\vdash_{\mathcal{S}} v_1^i : [\tau']$. We must have the derivation for all $i \in \{1..l\}$:

  $$\mathcal{S}_{i,1} = \frac{\vdash_{\mathcal{S}} v_1'^i : \tau' \quad \cdots \quad \vdash_{\mathcal{S}} v_{l_i}'^i : \tau'}{\vdash_{\mathcal{S}} [v_1'^i, \ldots, v_{l_i}'^i] : [\tau']}$$

  so $v_1^i = [v_1'^i, \ldots, v_{l_i}'^i]$.
  From the definition of $\langle \cdot \rangle$ we have

  $$\begin{array}{rl} & \langle v_1^1, \ldots, v_1^l \rangle_l \\ = & \langle [v_1'^1, \ldots, v_{l_1}'^1], \ldots, [v_1'^l, \ldots, v_{l_l}'^l] \rangle_l \\ = & ([l_1, \ldots, l_l], \langle ... \rangle_{l'}) \end{array}$$

  so $|c_1| = ([l_1, \ldots, l_l], \langle ... \rangle_{l'})$.
  From the definition of $|\cdot|$ we have

  $$([l_1, \ldots, l_l], \langle ... \rangle_{l'}) \quad = \quad |((..., [l_1, \ldots, l_l]), ...)|$$

  so $c_1 = ((..., [l_1, \ldots, l_l]), ...)$. By definition of *fst* and *snd* we have

  $$\mathit{snd}(\mathit{fst}(c_1)) = [l_1, \ldots, l_l]$$

  We have the evaluation

  $$\mathcal{O}\,[\![\textbf{length}]\!]\,(v_1^i) = \mathsf{R}_1\ \#v_1^i = \mathsf{R}_1\ l_i$$

  We have the vectorization

  $$\mathcal{O}\langle\!\langle \textbf{length} \rangle\!\rangle(\_x_1) = \textbf{lens}(\textbf{seg}(\_x_1))$$

  We have a vectorized evaluation

  $$\mathcal{T}_{\mathrm{s}}\,[\![\textbf{lens}(\textbf{seg}(\_x_1))]\!]\,\phi[\_x_1 \mapsto c_1] = \mathsf{R}_1\ \mathit{snd}(\mathit{fst}(c_1)) = \mathsf{R}_1\ [l_1, \ldots, l_l]$$

  We have value preservation by

  $$\begin{array}{rcl} |[l_1, \ldots, l_l]| & = & [l_1, \ldots, l_l] \\ & = & \langle l_1, \ldots, l_l \rangle_l \end{array}$$

  and work preservation by

  $$\begin{array}{rcl} 1 & \leq & K \cdot \sum_{i=1}^{l} 1 \\ & = & K \cdot l \end{array}$$

  for any $K \geq \frac{1}{l}$. Since $l \geq 1$ we can simply take $K = 1$.

- Case $o = \mathbf{conc}$:

  We have the type

  $$\mathbf{conc}[\tau'] : [[\tau']] \text{ −> } [\tau']$$

- Case $o = \mathbf{part}$:

  We have the type

  $$\mathbf{part}[\tau'] : [\tau'] * [\text{int}] \text{ −> } [[\tau']]$$

- Case $o = \mathbf{elt}$:

  We have the type

  $$\mathbf{elt}[\tau'] : [\tau'] * \text{int} \text{ −> } \tau'$$

  so $k = 2$ and $\tau_1 = [\tau']$, $\tau_2 = \text{int}$ and $\tau = \tau'$.

  Therefore we must have the derivation of

  $$\mathcal{S}_{i,1} = \frac{\vdash_{\mathcal{S}} v_1'^i : \tau' \quad \cdots \quad \vdash_{\mathcal{S}} v_{l_i}'^i : \tau'}{\vdash_{\mathcal{S}} [v_1'^i, \ldots, v_{l_i}'^i] : [\tau']}$$

  and

  $$\mathcal{S}_{i,2} = \frac{}{\vdash_{\mathcal{S}} n_i : \text{int}}$$

  for all $i \in \{1..l\}$. So $v_1^i = [v_1'^i, \ldots, v_{l_i}'^i]$ and $v_2^i = n_i$.

  From the definition of $\langle \cdot \rangle$ we have

  $$\begin{array}{rl} & \langle v_1^1, \ldots, v_1^l \rangle_l \\ = & \langle [v_1'^1, \ldots, v_{l_1}'^1], \ldots, [v_1'^l, \ldots, v_{l_l}'^l] \rangle_l \\ = & ([l_1, \ldots, l_l], \langle v_1'^1, \ldots, v_{l_l}'^l \rangle_{l'}) \end{array}$$

  so $|c_1| = ([l_1, \ldots, l_l], \langle v_1'^1, \ldots, v_{l_l}'^l \rangle_{l'})$, and

  $$c_1 = ([n_1', \ldots, n_l'], [l_1, \ldots, l_l], c_1')$$

  for some unknown starts $[n_1', \ldots, n_l']$ and value $c_1'$.

  Also

  $$c_2 = [n_1, \ldots, n_l]$$

  We have the evaluations

  $$\begin{array}{rcll} \mathcal{O}\,[\![\mathbf{elt}]\!]\,([v_1'^i, \ldots, v_{l_i}'^i], n_i) & = & \textit{if} & 0 \le n_i \le l_i - 1 \\ & & \textit{then} & \mathsf{R}_1\ v_{n_i-1}'^i \\ & & \textit{else} & \mathsf{E} \end{array}$$

We have the vectorization

$$\mathcal{O}\langle\!\langle \mathbf{elt} \rangle\!\rangle(\_x_1, \_x_2) = \begin{array}{ll} \mathbf{let} & \_x_1' \Leftarrow \_x_2 + \mathbf{starts}(\mathbf{seg}(\_x_1)) \\ & \_x_2' \Leftarrow [1] \\ \mathbf{in} & \mathbf{segfetch}(\mathbf{data}(\_x_1), \_x_1', \_x_2') \end{array}$$

We have the vectorized evaluation

$$\mathcal{T}_{\mathrm{s}} [\![\mathbf{elt}(\_x_1, \_x_2)]\!]_{=} \phi[\_x_1 \mapsto c_1, \_x_2 \mapsto c_2] \quad \begin{array}{ll} let^{\$} & c_1' \leftarrow \_x_2 + \mathbf{starts}(\mathbf{seg}(\_x_1)) \\ & c_2' \leftarrow [1] \\ in & \mathbf{segfetch}(\mathbf{data}(\_x_1), c_1', c_2') \end{array}$$

where

$$\begin{array}{lcl} \mathcal{T}_{\mathrm{s}} [\![\_x_2 + \mathbf{starts}(\mathbf{seg}(\_x_1))]\!]_l \, \phi[\_x_1 \mapsto c_1, \_x_2 \mapsto c_2] & = & \mathsf{R}_l \ [n_1 + n_1', \ldots, n_l + n_l'] \\ \mathcal{T}_{\mathrm{s}} [\![[1]]\!]_l \, \phi[\_x_1 \mapsto c_1, \_x_2 \mapsto c_2] & = & \mathsf{R}_l \ l \bullet 1 \\ \mathcal{T}_{\mathrm{s}} [\![\mathbf{segfetch}(\mathbf{data}(\_x_1), c_1', c_2')]\!]_l \, \phi[\_x_1 \mapsto c_1, \_x_2 \mapsto c_2] & = & \mathsf{R}_w \ c \end{array}$$

where $c = \mathit{segfetch}(c_1, [n_1 + n_1', \ldots, n_l + n_l'], l \bullet 1)$ and $w \leq 1 + 8l$ (from the work complexity of *segfetch* and $\sum_{i=1}^{l} 1 = l$).

We therefore have that the total work in vectorized evaluation is no more than $1 + 10l$. Assuming $l \geq 0$ we have work preservation for any $K \geq 11$.

Value preservation is not proved.

- Case $o = \mathbf{vec}_k$:

  We have the type

  $$\mathbf{vec}_k[\tau'] : {\tau'}^k \; {-}{>} \; [\tau'] \quad \mathit{for\ every}\ k \geq 0$$

□

**Theorem 14** *Work and value Preservation.*

$\forall e. \exists K. \forall \Gamma. \forall \tau. \forall l > 0. \forall \rho_1..\rho_l.$

$$\begin{array}{ll} \mathit{if} & \overset{\mathcal{S}}{\Gamma \vdash_{\mathcal{S}} e : \tau} \quad \mathit{and} \quad \overset{\mathcal{S}_1}{\vdash_{\mathcal{S}} \rho_1 : \Gamma} \quad \cdots \quad \overset{\mathcal{S}_l}{\vdash_{\mathcal{S}} \rho_l : \Gamma} \\ \\ \mathit{then}\ \forall \phi.\ \mathit{s.t.} & |\phi \restriction \mathit{dom}(\Gamma)| = \langle \rho_1, \ldots, \rho_l \rangle_l \\ \\ \mathit{we\ have} & \left| \mathcal{T}_{\mathrm{s}} [\![\langle\!\langle e \rangle\!\rangle]\!]_l \, \phi \right|^{\$} \lesssim_K \langle \mathcal{S} [\![e]\!] \rho_1, \ldots, \mathcal{S} [\![e]\!] \rho_l \rangle_l^{\$} \end{array}$$

**Proof** By lemma 2 on $\Gamma \vdash_{\mathcal{S}} e : \tau$ and $\vdash_{\mathcal{S}} \rho_1 : \Gamma \quad \cdots \quad \vdash_{\mathcal{S}} \rho_l : \Gamma$ we have an evaluation of

$$\mathcal{S}\,[\![e]\!]\,\rho_1 \qquad \cdots \qquad \mathcal{S}\,[\![e]\!]\,\rho_l$$

By lemma 10 on $\Gamma \vdash_{\mathcal{S}} e : \tau$ we have

$$\ll\Gamma\gg \vdash_{\mathcal{T}_{\mathrm{s}}} \langle\!\langle e \rangle\!\rangle : \ \ll\tau\gg$$

By lemma 11 on $\vdash_{\mathcal{S}} \rho_1 : \Gamma \quad \cdots \quad \vdash_{\mathcal{S}} \rho_l : \Gamma$ we have

$$\ll\Gamma\gg^{+} \vdash_{\mathcal{T}_{\mathrm{s}}} \left\langle \rho_1, \ldots, \rho_l \right\rangle_l :$$

So by lemma 3 and lemma 4 on $|\phi \restriction dom(\Gamma)| = \left\langle \rho_1, \ldots, \rho_l \right\rangle_l$ we have

$$\vdash_{\mathcal{T}} \phi \restriction dom(\Gamma) : \ \ll\Gamma\gg$$

Thus by lemma 6 we have an evaluation of

$$\mathcal{T}_{\mathrm{s}}\,[\![\langle\!\langle e \rangle\!\rangle]\!]_l\,\phi$$

The rest of the proof proceeds by induction on the syntax of $e$:

- Case $e = \overline{r}$:

  We have the evaluations

$$\begin{array}{rcl} \mathcal{S}\,[\![\overline{r}]\!]\,\rho_1 & = & \mathsf{R}_1\ r \\ & \vdots & \\ \mathcal{S}\,[\![\overline{r}]\!]\,\rho_k & = & \mathsf{R}_1\ r \end{array}$$

  We have the vectorization

$$\langle\!\langle \overline{r} \rangle\!\rangle = [\overline{r}]$$

  We have the vectorized evaluation

$$\mathcal{T}_{\mathrm{s}}\,[\![[\overline{r}]]\!]_l\,\phi = \mathsf{R}_l\ [\overbrace{r, \ldots, r}^{l}]$$

  We have value preservation by

$$\begin{array}{rcl} \left| [\overbrace{r, \ldots, r}^{l}] \right| & = & [\overbrace{r, \ldots, r}^{l}] \\ & = & \left\langle r, \ldots, r \right\rangle_l \end{array}$$

  We have work preservation by

$$\begin{array}{rcl} l & \leq & K \cdot \sum_{i=1}^{l} 1 \\ & = & K \cdot l \end{array}$$

  for any $K \geq 1$.

- Case $e = x$:
  From type soundness we have $x \in dom(\Gamma)$
  We have the evaluations

$$\mathcal{S}\,[\![x]\!]\,\rho_1 = \mathsf{R}_1\ \rho_1(x)$$

$$\vdots$$

$$\mathcal{S}\,[\![x]\!]\,\rho_l = \mathsf{R}_1\ \rho_l(x)$$

  We have the vectorization

$$\langle\!\langle x \rangle\!\rangle = x$$

  We have the vectorized evaluation

$$\mathcal{T}_{\mathrm{s}}\,[\![x]\!]_l\,\phi = \mathsf{R}_1\ \phi(x)$$

  We have value preservation by

$$\begin{array}{rcl} |\phi(x)| & = & |\phi|\,(x) \\ & \overset{x \in dom(\Gamma)}{=} & |\phi \upharpoonright dom(\Gamma)|\,(x) \\ & = & \langle \rho_1, \ldots, \rho_l \rangle_l (x) \\ & = & \langle \rho_1(x), \ldots, \rho_l(x) \rangle_l \\ & = & \langle v_1, \ldots, v_l \rangle_l \end{array}$$

  We have work preservation by

$$\begin{array}{rcl} 1 & \leq & K \cdot \displaystyle\sum_{i=1}^{l} 1 \\ & = & K \cdot l \end{array}$$

  for any $K \geq 1$.

- Case $e = \textbf{let}\ x \Leftarrow e_1\ \textbf{in}\ e_2$:

  By (TLet-T) we must have a derivation of

$$\dfrac{\begin{array}{cc} \mathcal{S}_1' & \mathcal{S}_2' \\ \Gamma \vdash_{\mathcal{S}} e_1 : \tau_1 & \Gamma[x \mapsto \tau_1] \vdash_{\mathcal{S}} e_2 : \tau_2 \end{array}}{\Gamma \vdash_{\mathcal{S}} \textbf{let}\ x \Leftarrow e_1\ \textbf{in}\ e_2 : \tau_2}$$

  We have the evaluations

$$\begin{array}{rll} \mathcal{S}\,[\![e]\!]\,\rho_i = & let^{\$} & v_1^i \leftarrow \mathcal{S}\,[\![e_1]\!]\,\rho_i \\ & in & \mathcal{S}\,[\![e_2]\!]\,\rho_i[x \mapsto v_1^i] \end{array}$$

for all $i \in \{1..l\}$. We have the vectorization

$$\langle\!\langle \textbf{let } x \Leftarrow e_1 \textbf{ in } e_2 \rangle\!\rangle = \textbf{let } x \Leftarrow \langle\!\langle e_1 \rangle\!\rangle \textbf{ in } \langle\!\langle e_2 \rangle\!\rangle$$

We have the vectorized evaluation

$$\begin{array}{rll} \mathcal{T}_{\mathrm{s}} \left[\!\left[ \textbf{let } x \Leftarrow \langle\!\langle e_1 \rangle\!\rangle \textbf{ in } \langle\!\langle e_2 \rangle\!\rangle \right]\!\right]_l \phi = & \mathit{let}^{\$} & c_1 \leftarrow \mathcal{T}_{\mathrm{s}} \left[\!\left[ \langle\!\langle e_1 \rangle\!\rangle \right]\!\right]_l \phi \\ & \mathit{in} & \mathcal{T}_{\mathrm{s}} \left[\!\left[ \langle\!\langle e_2 \rangle\!\rangle \right]\!\right]_l \phi[x \mapsto c_1] \end{array}$$

By IH on $\mathcal{S}_1'$ with $\mathcal{S}_1 \cdots \mathcal{S}_l$ we have

$$\left| \mathcal{T}_{\mathrm{s}} \left[\!\left[ \langle\!\langle e_1 \rangle\!\rangle \right]\!\right]_l \phi \right|^{\$} \lesssim_{K_1} \left\langle \mathcal{S} \left[\!\left[ e_1 \right]\!\right] \rho_1, \ldots, \mathcal{S} \left[\!\left[ e_1 \right]\!\right] \rho_l \right\rangle_l^{\$} \tag{1}$$

so $|c_1| = \left\langle v_1^1, \ldots, v_1^l \right\rangle_l$.
By lemma 2 on $\mathcal{S}_1'$ with $\vdash_{\mathcal{S}} \rho_i : \Gamma$ we have

$$\vdash_{\mathcal{S}} v_1^i : \tau_1 \quad \mathit{or} \quad \mathcal{S} \left[\!\left[ e_1 \right]\!\right] \rho_i = \mathsf{E}$$

for all $i \in \{1..l\}$.
If there exists an $i$ such that $\mathcal{S} \left[\!\left[ e_1 \right]\!\right] \rho_i = \mathsf{E}$, then

$$\left\langle \mathcal{S} \left[\!\left[ e_1 \right]\!\right] \rho_1, \ldots, \mathcal{S} \left[\!\left[ e_1 \right]\!\right] \rho_l \right\rangle_l^{\$} = \mathsf{E}$$

and by (1) we have

$$\mathcal{T}_{\mathrm{s}} \left[\!\left[ \langle\!\langle e_1 \rangle\!\rangle \right]\!\right]_l \phi = \mathsf{E}$$

so

$$\mathcal{T}_{\mathrm{s}} \left[\!\left[ \langle\!\langle e \rangle\!\rangle \right]\!\right]_l \phi = \mathsf{E}$$

Furthermore

$$\mathcal{S} \left[\!\left[ e \right]\!\right] \rho_i = \mathsf{E}$$

so

$$\left\langle \mathcal{S} \left[\!\left[ e \right]\!\right] \rho_1, \ldots, \mathcal{S} \left[\!\left[ e \right]\!\right] \rho_l \right\rangle_l^{\$} = \mathsf{E}$$

so we have congruence for any $K$.
Otherwise by (SEnvUp-T) we have the derivations of

$$\mathcal{S}_i'' = \frac{\vdash_{\mathcal{S}} \rho_i : \Gamma \qquad\qquad \vdash_{\mathcal{S}} v_1^i : \tau_1}{\vdash_{\mathcal{S}} \rho_i[x \mapsto v_1^i] : \Gamma[x \mapsto \tau_1]}$$

By IH on $\mathcal{S}_2'$ with $\mathcal{S}_1'' \quad \cdots \quad \mathcal{S}_l''$ we have

$$\forall \phi'.\ s.t.\ |\phi' \restriction \mathit{dom}(\Gamma[x \mapsto \tau_1])| = \left\langle \rho_1[x \mapsto v_1^1], \ldots, \rho_l[x \mapsto v_1^l] \right\rangle_l$$

$$\left|\mathcal{T}_{\mathrm{s}}\left[\!\left[\left\langle\!\left\langle e_2\right\rangle\!\right\rangle\right]\!\right]_l \phi'\right|^{\$} \lesssim_{K_2} \left\langle \mathcal{S}\left[\!\left[e_2\right]\!\right]\rho_1[x\mapsto v_1^1],\ldots,\mathcal{S}\left[\!\left[e_2\right]\!\right]\rho_l[x\mapsto v_1^l]\right\rangle_l^{\$} \quad (2)$$

We see that $\phi[x\mapsto c_0]$ satisfies this condition by

$$\begin{array}{rcl}
|\phi[x\mapsto c_0]\restriction dom(\Gamma[x\mapsto\tau_1])| & = & |\phi\restriction dom(\Gamma[x\mapsto\tau_1])[x\mapsto c_0]| \\
& = & |\phi\restriction dom(\Gamma[x\mapsto\tau_1])|\,[x\mapsto|c_0|] \\
& = & |\phi\restriction dom(\Gamma)|\,[x\mapsto|c_0|] \\
& = & \langle\rho_1,\ldots,\rho_l\rangle_l[x\mapsto|c_0|] \\
& = & \langle\rho_1,\ldots,\rho_l\rangle_l[x\mapsto\langle v_1^1,\ldots,v_1^l\rangle_l] \\
& = & \langle\rho_1[x\mapsto v_1^1],\ldots,\rho_l[x\mapsto v_1^l]\rangle_l
\end{array}$$

so we take $\phi'=\phi[x\mapsto c_0]$.

We have value preservation from (1) and we have work preservation from (1) and (2) for any $K\geq\max(K_1,K_2)$.

- Case $e=o(e_1,\ldots,e_k)$: We must have the type

$$\frac{\begin{array}{ccc}\mathcal{S}_1' & & \mathcal{S}_k' \\ \Gamma\vdash_{\mathcal{S}} e_1:\tau_1 & \cdots & \Gamma\vdash_{\mathcal{S}} e_k:\tau_k\end{array}}{\Gamma\vdash_{\mathcal{S}} o(e_1,\ldots,e_k):\tau}\,(o[\tau_1',\ldots,\tau_{k'}']:\tau_1*\cdots*\tau_k \text{ -> } \tau)$$

We have the evaluations

$$\begin{array}{rll}
\mathcal{S}\left[\!\left[o(e_1,\ldots,e_k)\right]\!\right]\rho_i = & let^{\$} & v_1^i\leftarrow\mathcal{S}\left[\!\left[e_1\right]\!\right]\rho_i \\
& & \vdots \\
& & v_k^i\leftarrow\mathcal{S}\left[\!\left[e_k\right]\!\right]\rho_i \\
& in & \mathcal{O}\left[\!\left[o\right]\!\right](v_1^i,\ldots,v_k^i)
\end{array}$$

for all $i\in\{1..l\}$. We have the Vectorization

$$\begin{array}{rll}
\left\langle\!\left\langle o(e_1,\ldots,e_k)\right\rangle\!\right\rangle = & \mathbf{let} & _x_1 \Leftarrow \left\langle\!\left\langle e_1\right\rangle\!\right\rangle \\
& & \vdots \\
& & _x_k \Leftarrow \left\langle\!\left\langle e_k\right\rangle\!\right\rangle \\
& \mathbf{in} & \mathcal{O}\left\langle\!\left\langle o\right\rangle\!\right\rangle(_x_1,\ldots,_x_k)
\end{array}$$

We have the vectorized evaluation

$$\begin{array}{rll}
\mathcal{T}_{\mathrm{s}}\left[\!\left[\left\langle\!\left\langle o(e_1,\ldots,e_k)\right\rangle\!\right\rangle\right]\!\right]_l\phi = & let^{\$} & c_1\leftarrow\mathcal{T}_{\mathrm{s}}\left[\!\left[\left\langle\!\left\langle e_1\right\rangle\!\right\rangle\right]\!\right]_l\phi \\
& & \vdots \\
& & c_k\leftarrow\mathcal{T}_{\mathrm{s}}\left[\!\left[\left\langle\!\left\langle e_k\right\rangle\!\right\rangle\right]\!\right]_l\phi \\
& in & \mathcal{T}_{\mathrm{s}}\left[\!\left[\mathcal{O}\left\langle\!\left\langle o\right\rangle\!\right\rangle(_x_1,\ldots,_x_k)\right]\!\right]_l\phi[_x_1\mapsto c_1,\ldots,_x_k\mapsto c_k]
\end{array}$$

By IH on $\mathcal{S}_i'$ with $\mathcal{S}_1\cdots\mathcal{S}_k$ we have

$$\left|\mathcal{T}_{\mathrm{s}}\left[\!\left[\left\langle\!\left\langle e_i\right\rangle\!\right\rangle\right]\!\right]_l \phi\right|^{\$} \lesssim_{K_i} \left\langle \mathcal{S}\left[\!\left[e_i\right]\!\right]\rho_1,\ldots,\mathcal{S}\left[\!\left[e_i\right]\!\right]\rho_l\right\rangle_l^{\$} \quad (1)$$

By lemma 2 on $\mathcal{S}'_1 \cdots \mathcal{S}'_k$ with $\vdash_{\mathcal{S}} \rho_i : \Gamma$ we have

$$\vdash_{\mathcal{S}} v_1^i : \tau_1 \quad or \quad \mathcal{S}\,[\![e_1]\!]\,\rho_i = \mathsf{E}$$

$$\vdots$$

$$\vdash_{\mathcal{S}} v_k^i : \tau_k \quad or \quad \mathcal{S}\,[\![e_k]\!]\,\rho_i = \mathsf{E}$$

for all $i \in \{1..l\}$.

Similar to the previous case, we can easily show congruence if there exists an $i \in \{1..l\}$ and a $j \in \{1..k\}$ such that $\mathcal{S}\,[\![e_j]\!]\,\rho_i = \mathsf{E}$.

Otherwise we can use lemma 13 on (1) and $\mathcal{S}'_1 \cdots \mathcal{S}'_k$ and $o[\tau'_1, \ldots, \tau'_{k'}] : \tau_1 * \cdots * \tau_k \;\text{->}\; \tau$ to get

$$\begin{aligned} &\mathcal{T}_{\mathrm{s}}\,\left[\!\left[\mathcal{O}\langle\!\langle o\rangle\!\rangle(\_x_1, \ldots, \_x_k)\right]\!\right]_l\,[\_x_1 \mapsto c_1, \ldots, \_x_k \mapsto c_k]^{\$} \\ &\qquad \lesssim_{K'} \left\langle \mathcal{O}\,[\![o]\!]\,(v_1^1, \ldots, v_k^1), \ldots, \mathcal{O}\,[\![o]\!]\,(v_1^l, \ldots, v_k^l)\right\rangle_l^{\$} \end{aligned} \tag{2}$$

We have value preservation from (2).

We have work preservation from (1) and (2) for any $K \geq \max(K', K_1, \ldots, K_k)$

- Case $e = [e_0 : x \in \mathbf{ix}(e_1)]$: By (SAPP-T) we must have a derivation of

$$\mathcal{S} = \frac{\begin{array}{cc} \mathcal{S}'_1 & \mathcal{S}'_0 \\ \Gamma \vdash_{\mathcal{S}} e_1 : \mathrm{int} & \Gamma[x \mapsto \mathrm{int}] \vdash_{\mathcal{S}} e_0 : \tau' \end{array}}{\Gamma \vdash_{\mathcal{S}} [e_0 : x \in \mathbf{ix}(e_1)] : [\tau']}$$

So $\tau = [\tau']$.

We have the evaluations

$$\mathcal{S}\,[\![[e_0 : x \in \mathbf{ix}(e_1)]]\!]\,\rho_i = \begin{array}{lll} let^{\$} & \multicolumn{2}{l}{n_i \leftarrow \mathcal{S}\,[\![e_1]\!]\,\rho_i} \\ in & \multicolumn{2}{l}{if\;\; n_i \geq 0} \\ & then & v_1^i \leftarrow \mathcal{S}\,[\![e_0]\!]\,\rho_i[x \mapsto 0] \\ & & \vdots \\ & & v_n^i \leftarrow \mathcal{S}\,[\![e_0]\!]\,\rho_i[x \mapsto n_i - 1] \\ & & \mathsf{R}_{n_i+1}\,[v_1^i, \ldots, v_n^i] \\ & else & \mathsf{E} \end{array}$$

for $i \in \{1..l\}$.

We have the vectorization

$$\langle\!\langle [e_0 : x \in \mathbf{ix}(e_1)] \rangle\!\rangle = \quad \begin{array}{ll} \mathbf{let} & \_x_1 \Leftarrow \langle\!\langle e_1 \rangle\!\rangle \\ & x \Leftarrow \mathbf{iotas}(\_x_1) \\ & \_x_2 \Leftarrow \mathbf{mkseg}(\_x_1) \\ & \_x_3 \Leftarrow \mathbf{par\ length}(x)\ \mathbf{do}\ lift_{\_x_1}^{FV(e_0)\setminus\{x\}}(\langle\!\langle e_0 \rangle\!\rangle) \\ \mathbf{in} & \mathbf{attsd}(\_x_2, \_x_3) \end{array}$$

We have the vectorized evaluation

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{T}_{\mathrm{s}} \left[\!\left[ \left\langle\!\left\langle [e_0 : x \in \mathbf{ix}(e_1)] \right\rangle\!\right\rangle \right]\!\right]_l \phi =} \\
\quad \mathit{let}^{\$} & c_1 \leftarrow \mathcal{T}_{\mathrm{s}} \left[\!\left[ \left\langle\!\left\langle e_1 \right\rangle\!\right\rangle \right]\!\right]_l \phi \\
& c_x \leftarrow \mathcal{T}_{\mathrm{s}} [\![\mathbf{iotas}(\_x_1)]\!]_l \, \phi[\_x_1 \mapsto c_1] \\
& c_2 \leftarrow \mathcal{T}_{\mathrm{s}} [\![\mathbf{mkseg}(\_x_1)]\!]_l \, \phi[\_x_1 \mapsto c_1, x \mapsto c_x] \\
& c_3 \leftarrow \mathcal{T}_{\mathrm{s}} \left[\!\left[ \mathbf{par\ length}(x)\ \mathbf{do}\ \mathit{lift}_{\_x_1}^{FV(e_0)\setminus\{x\}}(\left\langle\!\left\langle e_0 \right\rangle\!\right\rangle) \right]\!\right]_l \phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2] \\
\quad \mathit{in} & \mathcal{T}_{\mathrm{s}} [\![\mathbf{attsd}(\_x_2, \_x_3)]\!]_l \, \phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2, \_x_3 \mapsto c_3]
\end{array}
$$

- Sub-case $\left\langle \mathcal{S}[\![e]\!]\rho_1, \ldots, \mathcal{S}[\![e]\!]\rho_l \right\rangle_l^{\$} = \mathsf{E}$: In this case one of the following 3 conditions must be true by the evaluations of $e$. In all cases the vectorized evaluation fails as well. The point of failure is given informally:
  1. $\exists i \in \{1..l\}.\mathcal{S}[\![e_1]\!]\rho_i = \mathsf{E}$: Here $\left\langle\!\left\langle e_1 \right\rangle\!\right\rangle$ will fail by IH on $\mathcal{S}_1'$.
  2. $\exists i \in \{1..l\}.n_i < 0$: Here $\mathbf{iotas}(\_x_1)$ will fail.
  3. $\exists i \in \{1..l\}.\exists j \in \{0..n_i - 1\}.\mathcal{S}[\![e_0]\!]\rho_i[x \mapsto j] = \mathsf{E}$: Here $\left\langle\!\left\langle e_0 \right\rangle\!\right\rangle$ will fail by IH on $\mathcal{S}_0'$.

  In all cases $\mathcal{T}_{\mathrm{s}} \left[\!\left[ \left\langle\!\left\langle [e_0 : x \in \mathbf{ix}(e_1)] \right\rangle\!\right\rangle \right]\!\right]_l \phi = \mathsf{E}$, so

  $$\left| \mathcal{T}_{\mathrm{s}} \left[\!\left[ \left\langle\!\left\langle e \right\rangle\!\right\rangle \right]\!\right]_l \phi \right|^{\$} \lesssim_K \left\langle \mathcal{S}[\![e]\!]\rho_1, \ldots, \mathcal{S}[\![e]\!]\rho_l \right\rangle_l^{\$}$$

  for any $K$.

- Sub-case $\left\langle \mathcal{S}[\![e]\!]\rho_1, \ldots, \mathcal{S}[\![e]\!]\rho_l \right\rangle_l^{\$} = \mathsf{R}_w\ c$: Conversely to the last case all of these conditions must be met:
  1. $\forall i \in \{1..l\}.\mathcal{S}[\![e_1]\!]\rho_i = \mathsf{R}_{w_1^i}\ n_i$
  2. $\forall i \in \{1..l\}.n_i \geq 0$
  3. $\forall i \in \{1..l\}.\forall j \in \{0..n_i - 1\}.\mathcal{S}[\![e_0]\!]\rho_i[x \mapsto j] = \mathsf{R}_{w_0^{i,j}}\ v_j^i$

     Furthermore the following must be true by the evaluations of $e$ and the definition of $\left\langle \cdot \right\rangle^{\$}$
  4. $w = \sum w_1^i + \sum w_0^{i,j} + \sum (n_i + 1)$

  By IH on $\mathcal{S}_1'$ with $\mathcal{S}_1 \cdots \mathcal{S}_l$ we have

  $$\left| \mathcal{T}_{\mathrm{s}} \left[\!\left[ \left\langle\!\left\langle e_1 \right\rangle\!\right\rangle \right]\!\right]_l \phi \right|^{\$} \lesssim_{K_1} \left\langle \mathcal{S}[\![e_1]\!]\rho_1, \ldots, \mathcal{S}[\![e_1]\!]\rho_l \right\rangle_l^{\$} \tag{5}$$

  From (1) and (5) we have $\mathcal{T}_{\mathrm{s}} \left[\!\left[ \left\langle\!\left\langle e_1 \right\rangle\!\right\rangle \right]\!\right]_l \phi = \mathsf{R}_{w_1'}\ [n_1, \ldots, n_l]$ for some $w_1' \leq K_1 \cdot \sum_{i=1}^{l} w_1^i$, and $c_1 = [n_1, \ldots, n_l]$.
  Let $l' = \sum_{i=1}^{l} n_i$. We have

  $$\mathcal{T}_{\mathrm{s}} [\![\mathbf{iotas}(\_x_1)]\!]_l \, \phi[\_x_1 \mapsto c_1] \stackrel{(2)}{=} \mathsf{R}_{l+l'}\ \mathit{iotas}(c_1)$$

  so $c_x = \mathit{iotas}(c_1)$, also let $w_x' = l + l'$.

  $$\mathcal{T}_{\mathrm{s}} [\![\mathbf{mkseg}(\_x_1)]\!]_l \, \phi[\_x_1 \mapsto c_1, x \mapsto c_x] = \mathsf{R}_{1+l}\ (\mathit{scan}_+(c_1), c_1)$$

so $c_2 = (scan_+(c_1), c_1)$, $w'_2 = 1 + l$.
Assume $l' > 0$ (this sub-case also holds for $l' = 0$ but it will not be shown) we then have:

$$\mathcal{T}_\mathrm{s}\left[\!\!\left[\mathbf{par\ length}(x)\ \mathbf{do}\ lift^{FV(e_0)\setminus\{x\}}_{\_x_1}(\langle\!\langle e_0\rangle\!\rangle)\right]\!\!\right]_l \phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2]$$

$$\begin{array}{rll} = & \text{let} & l'' = \mathcal{U}\,[\![\mathbf{length}(x)]\!]\,\phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2] \\ & \text{in} & \mathcal{T}_\mathrm{s}\left[\!\!\left[lift^{FV(e_0)\setminus\{x\}}_{\_x_1}(\langle\!\langle e_0\rangle\!\rangle)\right]\!\!\right]_{l''} \phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2] \end{array}$$

$l''$ is evaluated to

$$\begin{array}{rcl} \mathcal{U}\,[\![\mathbf{length}(x)]\!]\,\phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2] & = & \#(c_x) \\ & = & \#(iotas(c_1)) \\ & = & l' \end{array}$$

So $l'' = l'$.
By lemma 12 on $\phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2](\_x_1) = [n_1, \ldots, n_l]$, (2) and the assumption $l' > 0$ we have

$$\mathcal{T}_\mathrm{s}\left[\!\!\left[lift^{FV(e_0)\setminus\{x\}}_{\_x_1}(\langle\!\langle e_0\rangle\!\rangle)\right]\!\!\right]_{l'} \phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2]$$

$$\begin{array}{rll} \lesssim & let^{\$} & \_ \leftarrow \mathtt{Tick}_{l+2l'+4k(1+l+l')} \\ & in & \mathcal{T}_\mathrm{s}\,[\![\langle\!\langle e_0\rangle\!\rangle]\!]_{l'}\, lift^{FV(e_0)\setminus\{x\}}_{c_1}(\phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2]) \end{array} \quad (6)$$

By (SEnvUp-T) for all $i \in \{1..l\}$ for all $j \in \{0..n_i - 1\}$ we have

$$\mathcal{S}'_{i,j} = \dfrac{\dfrac{\mathcal{S}_i}{\vdash_\mathcal{S} \rho_i : \Gamma} \qquad \dfrac{}{\vdash_\mathcal{S} j : \mathrm{int}}}{\vdash_\mathcal{S} \rho_i[x \mapsto j] : \Gamma[x \mapsto \mathrm{int}]}$$

By IH on $\mathcal{S}'_0$ and $\mathcal{S}'_{i,j}$ with $l'$ we have

$$\forall \phi'\ s.t.\ |\phi' \restriction dom(\Gamma[x \mapsto \mathrm{int}])|$$

$$= \left\langle \rho_1[x \mapsto 0], \ldots, \rho_1[x \mapsto n_1 - 1], \ldots, \rho_l[x \mapsto 0], \ldots, \rho_l[x \mapsto n_l - 1]\right\rangle_{l'}$$

we have

$$\begin{array}{l} \left|\mathcal{T}_\mathrm{s}\,[\![\langle\!\langle e_0\rangle\!\rangle]\!]_{l'}\,\phi'\right|^{\$} \lesssim_{K_0} \\ \langle \mathcal{S}\,[\![e_0]\!]\,\rho_1[x \mapsto 0], \ldots, \mathcal{S}\,[\![e_0]\!]\,\rho_1[x \mapsto n_1 - 1], \\ \quad \ldots, \mathcal{S}\,[\![e_0]\!]\,\rho_l[x \mapsto 0], \ldots, \mathcal{S}\,[\![e_0]\!]\,\rho_l[x \mapsto n_l - 1]\rangle^{\$}_{l'} \end{array} \quad (7)$$

Take $\phi' = lift_{c_1}^{dom(\phi)\setminus\{x\}}(\phi[x \mapsto c_x])$. It satisfies the condition by

$$
\begin{array}{cl}
 & |\phi' \restriction dom(\Gamma[x \mapsto \mathrm{int}])| \\
\overset{unfold\ \phi'}{=} & \left|lift_{[n_1,\ldots,n_l]}^{dom(\phi)\setminus\{x\}}(\phi[x \mapsto c_x]) \restriction dom(\Gamma[x \mapsto \mathrm{int}])\right| \\
\overset{x\notin dom(\phi)\setminus\{x\}}{=} & \left|lift_{[n_1,\ldots,n_l]}^{dom(\phi)\setminus\{x\}}(\phi)[x \mapsto c_x] \restriction dom(\Gamma[x \mapsto \mathrm{int}])\right| \\
\overset{x\in dom(\Gamma[x\mapsto \mathrm{int}])}{=} & \left|lift_{[n_1,\ldots,n_l]}^{dom(\phi)}(\phi \restriction dom(\Gamma))[x \mapsto c_x]\right| \\
\overset{c_x\ is\_flat}{=} & \left|lift_{[n_1,\ldots,n_l]}^{dom(\phi)}(\phi \restriction dom(\Gamma))\right| [x \mapsto c_x] \\
\overset{\phi\restriction dom(\Gamma)=\left\langle\rho_1,\ldots,\rho_l\right\rangle_l,(2)}{=} & \big\langle \overbrace{\rho_1,\ldots,\rho_1}^{n_1},\ldots,\overbrace{\rho_l,\ldots,\rho_l}^{n_l} \big\rangle_{l'}[x \mapsto c_x] \\
= & \big\langle \rho_1[x \mapsto 0],\ldots,\rho_1[x \mapsto n_1-1],\ldots,\rho_l[x \mapsto 0],\ldots,\rho_l[x \mapsto n_l-1]\big\rangle_{l'}
\end{array}
$$

From (3) and (7) we have

$$\mathcal{T}_{\mathsf{s}}\,[\![\langle\!\langle e_0\rangle\!\rangle]\!]_{l'}\,\phi' = \mathsf{R}_{w_0'}\ c_0 \tag{8}$$

for some $w_0' \leq \sum_{i=1}^{l}\sum_{j=1}^{l_i'} w_0^{i,j}$ and $|c_0| = \langle v_0^1,\ldots,v_{n_1-1}^1,\ldots,v_0^l,\ldots,v_{n_l-1}^l\rangle_{l'}$.
So by lemma 7, because $\_x_1$ and $\_x_2$ is selected to be non-free in $\langle\!\langle e_0\rangle\!\rangle$ and because $FV(e_0) = FV(\langle\!\langle e_0\rangle\!\rangle)$ we also have

$$\mathcal{T}_{\mathsf{s}}\,[\![\langle\!\langle e_0\rangle\!\rangle]\!]_{l'}\,lift_{c_1}^{FV(e_0)\setminus\{x\}}(\phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2]) = \mathsf{R}_{w_0'}\ c_0$$

So by (6) we have

$$\mathcal{T}_{\mathsf{s}}\left[\!\!\left[lift_{\_x_1}^{FV(e_0)\setminus\{x\}}(\langle\!\langle e_0\rangle\!\rangle)\right]\!\!\right]_{l'}\phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2] = \mathsf{R}_{w_0''}\ c_0$$

for some $w_0'' \leq w_0'$.
And by the vectorized evaluation of **par length**$(x)$ **do** $lift_{\_x_1}^{FV(e_0)\setminus\{x\}}(\langle\!\langle e_0\rangle\!\rangle)$ we have

$$\mathcal{T}_{\mathsf{s}}\left[\!\!\left[\textbf{par length}(x)\ \textbf{do}\ lift_{\_x_1}^{FV(e_0)\setminus\{x\}}(\langle\!\langle e_0\rangle\!\rangle)\right]\!\!\right]_{l}\phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2] = \mathsf{R}_{w_0''}\ c_0$$

We then have

$$\mathcal{T}_{\mathsf{s}}\,[\![\textbf{attsd}(\_x_2,\_x_3)]\!]_{l}\,\phi[\_x_1 \mapsto c_1, x \mapsto c_x, \_x_2 \mapsto c_2, \_x_3 \mapsto c_0] = \mathsf{R}_1\ (c_2, c_0)$$

so

$$\mathcal{T}_{\mathsf{s}}\,[\![\langle\!\langle [e_0 : x \in \mathbf{ix}(e_1)]\rangle\!\rangle]\!]_{l}\,\phi = \mathsf{R}_{w'}\ (c_2, c_0)$$

and $w' \leq w_1 + (l+l') + (1+l) + (l+2l'+4k(1+l+l')) + w_0''$ where

$k = |FV(e_0)\backslash\{x\}|$. Therefore the value is preserved by

$$
\begin{array}{lcl}
|c| & = & |(c_2, c_0)| \\
 & = & |((scan_+(c_1), c_1), c_0)| \\
 & \overset{def.\ of\ |\cdot|}{=} & (c_1, \triangle_{c_1}^{scan_+(c_1)}(c_0)) \\
 & \overset{Lemma\ 5}{=} & (c_1, |c_0|) \\
 & \overset{unfold\ c_1}{=} & ([n_1, \ldots, n_l], |c_0|) \\
 & \overset{(8)}{=} & ([n_1, \ldots, n_l], \langle v_0^1, \ldots, v_{n_1-1}^1, \ldots, v_0^l, \ldots, v_{n_l-1}^l \rangle_{l'}) \\
 & \overset{(3)}{=} & \langle \mathcal{S}\,[\![e]\!]\,\rho_1, \ldots, \mathcal{S}\,[\![e]\!]\,\rho_l \rangle_l
\end{array}
$$

The work is preserved by

$$
\begin{array}{lll}
w' & \leq & w_1' + (l + l') + (1 + l) + (l + 2l' + 4k(1 + l + l')) + w_0'' \\
 & \leq & 1 + 3l + 3l' + 4k(1 + l + l') + w_1' + w_0' \\
 & \leq & 1 + 3l + 3l' + 4k(1 + l + l') + K_1 \cdot \sum w_1^i + K_0 \cdot \sum w_0^{i,j} \\
 & \leq^{(*)} & K'' \cdot l + K'' \cdot l' + K_1 \cdot \sum w_1^i + K_0 \cdot \sum w_0^{i,j} \\
 & \leq & K \cdot (\sum w_1^i + \sum w_0^{i,j} + l' + l) \quad \left(K = \max(K_0, K_1, \frac{12k+7}{2})\right) \\
 & \overset{(4)}{=} & K \cdot w
\end{array}
$$

$(*)$: $1 + 3l + 3l' + 4k(1 + l + l') \leq K'' \cdot l + K'' \cdot l'$ has the solution $K'' \geq \frac{12k+7}{2}$.

□

**The Constant $K$** Based on the proof of theorem 14, it is possible to define the constant $K$ as a function of the expression.

$K : \mathbf{SExp} \rightarrow \mathrm{N}$

$$
\begin{array}{rcl}
K(\overline{d}) & = & 1 \\
K(x) & = & 1 \\
K(\mathbf{let}\ x \Leftarrow e_1\ \mathbf{in}\ e_2) & = & \max(K(e_1), K(e_2)) \\
K(o(e_1, \ldots, e_k)) & = & \max(K(e_1), \ldots, K(e_k), K(o)) \\
K([e_0 : x \in e_1]) & = & \max\left(K(e_0), K(e_1), \dfrac{12FV(e_0) + 7}{2}\right) \\
K(\mathbf{vec}_k) & = & K_{vec} \\
K(\mathbf{length}) & = & 1 \\
K(\mathbf{elt}) & = & 11 \\
K(\mathbf{conc}) & = & K_{conc} \\
K(\mathbf{part}) & = & K_{part} \\
K(\oplus) & = & 1
\end{array}
$$

Note that $K_{vec}$, $K_{conc}$ and $K_{part}$ have not been shown formally yet, but they are strongly believed to be constants.

# 4 Implementation

An interpreter for the source an target language has been implemented in Haskell. Apart from values, the interpreter also computes the work complexity as defined by the semantic functions. A parser for the source language has also been implemented, and finally the flattening transformation has been implemented.

The implementations of the interpreters are not ideal - they do not obey the cost models presented in this paper. This it not because it is not possible, it is simply because a cost-correct implementation is out of the scope of this report.

The implementation serves to support the value and work preservation proof as well as to demonstrate actual values of the constant $K$. Furthermore the implementation is also used as an informal argument why the vectorization is still work-efficient after including mutual recursive user-defined function by demonstrating work-efficiency on the Quicksort algorithm.

# 5 Evaluation

## 5.1 Replication Problem

Generalized segment descriptors has proven to be a useful tool to achieve theoretical better work complexities. The cost of replication of values required by the vectorization allows a reference-based cost model of the target language, which in turn allows a better cost model for the source language. Although we do have special transformation rules for each primitive operation, our solution is still more general than the predominant solution of treating indexing as a special case, since we still replicate all arguments indiscriminately.

In each apply-to-each construct, the work complexity of replication is now in order of the parallel degree times the number of free variables in the body of the apply-to-each.

The downside of our solution of the replication problem is that we increase the amount of read contention.

## 5.2 Provable Correct Cost Model

Our source language cost model is close to the ideal cost model with the exception of vector constructor, and our vectorization has been proved to preserved work complexity within a factor $K$ that depends on the number of free variables.

This $K$ could be made truly constant by incorporating the cost of replication in the source language cost model by charging for the replication in every apply-to-each. This would yield a slightly worse source language cost model where apply-to-each has and additional cost of the sum of the lengths of the free variables.

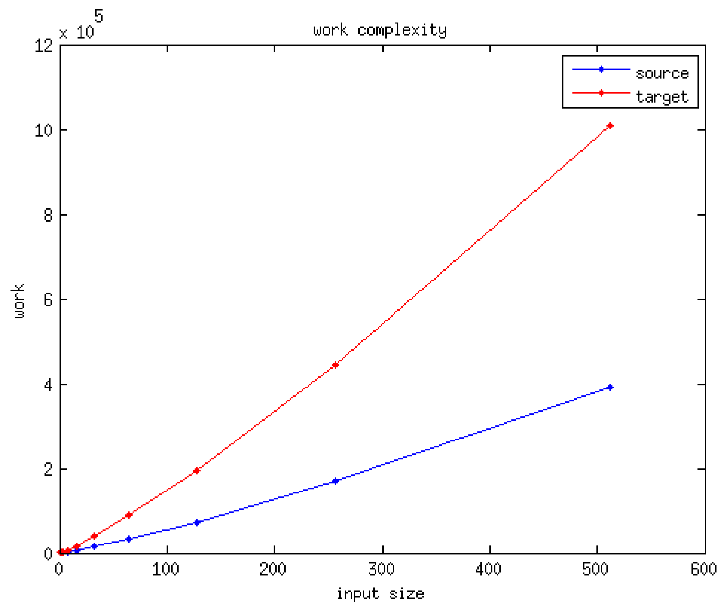
This cost model is still better than the construct-parameters cost model for Proteus where the apply-to-each charges and additional cost of the sum of the sizes of the free variables, because the size of a value is always greater than or equal to its length.

On the other hand our cost model is hard to compare to the construct-result cost model of Proteus where the apply-to-each charges an additional cost of the size of the evaluated expression. Whether the length of the values bound to the free variables or the size of the evaluated expression is greatest depends on the expression. But our vectorization does not pose restrictions on programs as does Proteus with construct-result semantics.

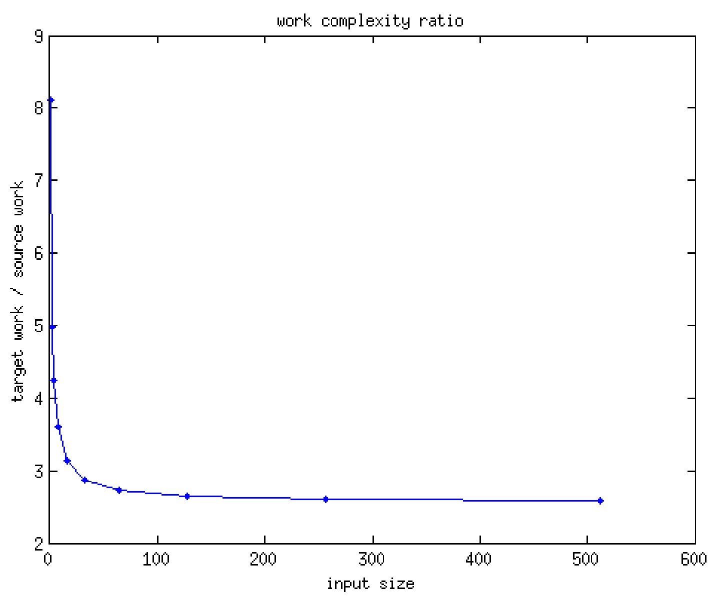
In conclusion our cost model is novel in that it provides better work complexities (with the exception of vector construction) than any existing provable and implementable cost model for programs without restrictions.

## 5.3 Implementation Results

The following figure shows the work complexity of Quicksort evaluated by the cost model for the source language and for the target language as presented in this report.




The next figure shows the ratio between the work complexity of the two languages. This would correspond to the constant $K$.




The target language is clearly more expensive than the source language, but not asymptotically, which is exactly the desired results. The value of $K$ seems to be large for very small input sizes, which is acceptable, but tend to stabilize around 2.8 for the Quicksort algorithm. A 2.8 times higher work-complexity may seem significant, but if the program is in turn able to run on 1000 parallel processes the actual wall-clock time may indeed by much smaller. Whether this is true or not depends on if the step complexity can be preserved and if the target language cost model is actually implementable. The answer to both of these questions are left for future work.

# 6 Conclusion

## 6.1 Future Work

There are still many unexplored aspects of this vectorization.

- Scattered Segments: To reduce the work complexity of the vector constructor to match the ideal semantics, we would generalize segment descriptors even further by allowing them to point to different data vectors.
- Step Complexity: Including the step complexity in the cost model will reveal how parallelizable the vectorization is. It is expected that only contained programs are depth preserving.
- Divergence: In order to formalize user-defined recursive functions, we would need to treat divergence, possibly using a formulation with CPOS or similar.
- Cost-correct implementation and empirical comparison to the existing solutions.
- Analysis of how much the problem of concurrent reads dominates the work complexity, and possibly a heuristic solution hereof.

We have presented a nested data parallel language, a flat data parallel language and a flattening transformation from the former to the latter. We have given a reasonable work complexity cost model for both languages, with the exception of the cost of the vector constructor in the source language, and we have proved that the flattening transformation preserves value and work. The work is preserved within a factor of $K$ for some $K$ that may depend on the size of the source language expression.

The cost model for the source language is ideal (except for vector construction) and the formally correct vectorization is therefore novel.

The formulation of the languages and the vectorization has been derived by studying and comparing existing vectorizations in NESL, Proteus and Data Parallel Haskell and the success of the work-efficiency can be largely attributed to the introduction of the key concept of generalized segment descriptors.

# References

[Blelloch(1995)] Guy Blelloch. Nesl: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, 1995. URL http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-95-170.ps.gz.

[Blelloch(1990)] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-262-02313-X.

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

[Blelloch and Sabot(1990)] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8:119–134, February 1990. ISSN 0743-7315. doi: 10.1016/0743-7315(90)90087-6. URL http://dl.acm.org/citation.cfm?id=78246.78250.

[Chakravarty(2011)] M. Chakravarty. Shared data structures in nested data parallelism. Presented at the 2nd HIPERFIT workshop, Copenhagen, December 2011. URL http://hiperfit.dk/pdf/HIPERFIT-2-chakravarty.pdf.

[Chakravarty et al.(2007)Chakravarty, Leshchinskiy, Jones, Keller, and Marlow] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. doi: http://doi.acm.org/10.1145/1248648.1248652. URL http://doi.acm.org/10.1145/1248648.1248652.

[Keller and Simons(1996)] Gabriele Keller and Martin Simons. A calculational approach to flattening nested data parallelism in functional languages. In *Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security*, ASIAN '96, pages 234–243, London, UK, 1996. Springer-Verlag. ISBN 3-540-62031-1. URL http://dl.acm.org/citation.cfm?id=646063.676331.

[Palmer et al.(1995)Palmer, Prins, and Westfold] D. W. Palmer, J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, FRONTIERS '95, pages 186–, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-6965-9. URL http://dl.acm.org/citation.cfm?id=528717.796649.

[Peyton Jones(2008)] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5. doi: http:

//dx.doi.org/10.1007/978-3-540-89330-1_10. URL http://dx.doi.org/10.1007/978-3-540-89330-1_10.

[Prins and Palmer(1993)] Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. In *IN PROCEEDINGS OF THE FOURTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING*, pages 119–128. ACM, 1993. URL http://doi.acm.org/10.1145/173284.155345.

[Riely et al.(1995)Riely, Prins, and Iyer] J. W. Riely, J. Prins, and S. P. Iyer. Provably correct vectorization of nested-parallel programs. In *Proceedings of the conference on Programming Models for Massively Parallel Computers*, PMMP '95, pages 213–, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7177-7. URL http://dl.acm.org/citation.cfm?id=525697.826731.

[Roman Leshchinskiy and Keller(2006)] Manuel M. T. Chakravarty Roman Leshchinskiy and Gabriele Keller. Higher order flattening. In *V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, ICCS'06: International Conference on Computational Science (2)*, pages 920–928. Springer-Verlag, 2006. URL http://www.cse.unsw.edu.au/~rl/publications/ho-flattening.ps.gz.

# Appendix A: All lemmas and theorems

**Lemma 1** *Typing Context Inclusion (source).*

$$\forall \Gamma, \Gamma'.$$

$$if\ \Gamma \subseteq \Gamma'\ and\ \Gamma \vdash_{\mathcal{S}} e : \tau\ then\ \Gamma' \vdash_{\mathcal{S}} e : \tau$$

**Lemma 2** *Type Soundness (source).*

$$if\ \Gamma \vdash_{\mathcal{S}} e : \tau\ and\ \vdash_{\mathcal{S}} \rho : \Gamma$$

$$then\quad either\quad \mathcal{S}\,[\![e]\!]\,\rho = \mathsf{E}$$

$$or\quad \mathcal{S}\,[\![e]\!]\,\rho = \mathsf{R}_w\ v\quad and\quad \vdash_{\mathcal{S}} v : \tau$$

**Lemma 3** *Type Context Inclusion (target).*

$$\forall \Pi, \Pi'.$$

$$if\ \Pi \subseteq \Pi'\ and\ \Pi \vdash_{\mathcal{T}_s} s : \sigma\ then\ \Pi' \vdash_{\mathcal{T}_s} s : \sigma$$

**Lemma 4** *Normalization type preservation.*

$$\vdash_{\mathcal{T}_s} c : \sigma$$

$$if\ and\ only\ if\ \vdash_{\mathcal{T}_s} |c| : |\sigma|$$

**Lemma 5**

$$if \ \sum_{i=1}^{l} n_i = \#c$$

$$then \ \triangle^{[n_1,\ldots,n_l]}_{scan_+([n_1,\ldots,n_l])}(c) = |c|$$

**Lemma 6** *Type Soundness (target.*

$$if \ \Pi \vdash_{\mathcal{T}_s} s : \sigma \ and \ \vdash_{\mathcal{T}} \phi : \Pi$$

$$then \quad either \quad \mathcal{T}_s [\![s]\!]_l \, \phi = \mathsf{E}$$

$$or \qquad \mathcal{T}_s [\![s]\!]_l \, \phi = \mathsf{R}_w \ c \quad and \quad \vdash_{\mathcal{T}_s} c : \sigma$$

**Lemma 7** *Restriction to free variables.*

$$\mathcal{T}_s [\![s]\!]_l \, \phi = \mathcal{T}_s [\![s]\!]_l \, \phi \restriction FV(s)$$

**Lemma 8** *Lifting expression type preservation.*

$$If \quad \Pi \vdash_{\mathcal{T}_s} s : \sigma \quad and \quad \Pi(x) = [int] \quad and \quad X \subseteq dom(\Pi) \quad and \quad x \notin X$$

$$then \quad \Pi \vdash_{\mathcal{T}_s} lift^X_x(s) : \sigma$$

**Lemma 9** *Operation Type Preservation.*

$$\forall o. \forall \tau'_1..\tau'_{k'}.$$

$$if \quad o[\tau'_1, \ldots, \tau'_{k'}] : \tau_1 * \cdots * \tau_k \ -> \tau$$

$$then \quad \forall \ \Pi. \quad \Pi[\_x_1 \mapsto \ll \tau_1 \gg, \ldots, \_x_k \mapsto \ll \tau_k \gg] \overset{\mathcal{T}}{\vdash_{\mathcal{T}_s}} \mathcal{O}\langle\!\langle o \rangle\!\rangle(\_x_1, \ldots, \_x_k) : \ \ll \tau \gg$$

**Lemma 10** *Expression Type Preservation.*

$$\forall e. \forall \Gamma.$$

$$if \quad \Gamma \vdash_{\mathcal{S}} e : \tau$$

$$then \quad \ll \Gamma \gg \vdash_{\mathcal{T}_s} \langle\!\langle e \rangle\!\rangle : \ \ll \tau \gg$$

**Lemma 11** *Value Type Preservation.*

$$if \qquad \overset{\mathcal{S}_1}{\vdash_{\mathcal{S}}} v_1 : \tau \quad \cdots \ and \ \cdots \quad \overset{\mathcal{S}_l}{\vdash_{\mathcal{S}}} v_l : \tau \qquad (l \geq 0)$$

$$then \qquad \overset{\mathcal{T}}{\vdash_{\mathcal{T}_s}} \langle v_1, \ldots, v_l \rangle_l : \ \ll \tau \gg^+$$

**Lemma 12** *Lifting value and work preservation.*

$$if\ \phi(x) = ixrep_l(n_1, \ldots, n_l) \quad and \quad \forall i.n_i \geq 0 \quad and \quad l' = \sum_{i=1}^{l} n_i > 0$$

$$then\ \mathcal{T}_{\mathrm{s}} \left[\!\left[ lift_x^X(t) \right]\!\right]_l \phi$$

$$\begin{array}{ll} \lesssim & let^{\$} \quad \_ \leftarrow \boldsymbol{\mathtt{Tick}}_{l+2l'+|X|(1+4l+4l')} \\ & in \quad\ \ \mathcal{T}_{\mathrm{s}} [\![t]\!]_{l'}\, lift^X_{[n_1,\ldots,n_l]}(\phi) \end{array}$$

**Lemma 13** *Operation work and value preservation.*

$$if\ \ |c_1| = \left\langle v_1^1, \ldots, v_1^l \right\rangle_l \quad \cdots \quad |c_k| = \left\langle v_k^1, \ldots, v_k^l \right\rangle_l$$

$$and\ \vdash_{\mathcal{S}} v_1^1 : \tau_1 \quad \cdots \quad \vdash_{\mathcal{S}} v_k^l : \tau_k$$

$$and\ o[\tau_1', \ldots, \tau_{k'}'] : \tau_1 * \cdots * \tau_k\ {-}{>}\ \tau$$

$$then\ \forall \phi. \left| \mathcal{T}_{\mathrm{s}} \left[\!\left[ \mathcal{O}\langle\!\langle o \rangle\!\rangle(\_x_1, \ldots, \_x_k) \right]\!\right]_l \phi[\_x_1 \mapsto c_1, \ldots, \_x_k \mapsto c_k] \right|^{\$}$$

$$\lesssim_K \left\langle \mathcal{O} [\![o]\!]\, (v_1^1, \ldots, v_k^1), \ldots, \mathcal{O} [\![o]\!]\, (v_1^l, \ldots, v_k^l) \right\rangle_l^{\$}$$

**Theorem 14** *Work and value Preservation.*

$$\forall e. \exists K. \forall \Gamma. \forall \tau. \forall l > 0. \forall \rho_1..\rho_l.$$

$$\begin{array}{lllll} if & \overset{\mathcal{S}}{\Gamma \vdash_{\mathcal{S}} e : \tau} & and\ \overset{\mathcal{S}_1}{\vdash_{\mathcal{S}} \rho_1 : \Gamma} & \cdots & \overset{\mathcal{S}_l}{\vdash_{\mathcal{S}} \rho_l : \Gamma} \end{array}$$

$$then\ \forall \phi.\ s.t. \qquad |\phi \restriction dom(\Gamma)| = \left\langle \rho_1, \ldots, \rho_l \right\rangle_l$$

$$we\ have \quad \left| \mathcal{T}_{\mathrm{s}} \left[\!\left[ \langle\!\langle e \rangle\!\rangle \right]\!\right]_l \phi \right|^{\$} \lesssim_K \left\langle \mathcal{S} [\![e]\!]\, \rho_1, \ldots, \mathcal{S} [\![e]\!]\, \rho_l \right\rangle_l^{\$}$$