Tensor Comprehensions in SaC

Sven-Bodo Scholz
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
s.scholz@hw.ac.uk

Artjoms Šinkarovs
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
a.sinkarovs@hw.ac.uk

ABSTRACT

We propose a new notation for data parallel operators on multi-dimensional arrays named *tensor comprehensions*. This notation combines the basic principle of array-comprehensions with syntactical shortcuts very close to those found in the so-called Tensor Notations used in Physics and Mathematics. As a result, complex operators with rich semantics can be defined concisely. The key to this conciseness lies in the ability to define shape-polymorphic operations combined with the ability to infer array shapes from the immediate context. The paper provides a definition of the proposed notation, a formal shape inference process, as well as a set of re-write rules that translates tensor comprehensions as a zero-cost syntactic sugar into standard SaC expressions.

ACM Reference Format:
Sven-Bodo Scholz and Artjoms Šinkarovs. 2020. Tensor Comprehensions in SaC. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Most array languages offer a variety of built-in operators that are commonly used in practice, *e.g.* linear algebra, tensor operations, *etc.* Typically, these operators are pure functions that can be easily composed, opening a great potential for program analysis and optimisations.

While these operations are very powerful when expressing homogeneous computations on entire arrays, they lead to rather cumbersome specifications when operating on subsets of the elements only; arrays have to be disassembled into sub-arrays before applying the desired functionality and re-assembled thereafter. As a simple example, consider incrementing all elements but those in the lowest and highest index positions of a vector v[1]:

```
   take([1 ], v)
++ drop([1 ], drop([-1], v)) + 1
++ take([-1], v)
```

[1]We use the array operators as they are provided in the SaC standard library; similar operators can be found in all array languages.

IFL'19, September 2019, Singapore

ACM ISBN 978-1-4503-7562-7…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Note here, that negative first arguments to the functions `take` and `drop` relate to elements from the "right", *i.e.* starting from the highest index.

If we apply this approach to a matrix we need to create 5 sub-arrays, only one of which actually is being incremented. Looking at the generic case of an array of rank n, we need to involve $2^n + 1$ sub-arrays. Such inconvenient specifications can be avoided through the introduction of array comprehensions. Array comprehensions introduce explicit references to indices, enabling the identification of the relevant elements through index relations. Using the array comprehensions of SaC, named *with-loop*s, the above example can now be specified as

```
with {
  ([1] <= [i] < shape(v)-1) : v[i] + 1;
} : modarray( v);
```

A further benefit of such comprehensions is the straight-forward extensibility to higher ranks. For matrices `m`, we can write

```
with {
  ([1,1] <= [i,j] < shape(m)-1) : m[i,j] + 1;
} : modarray( m);
```

and in shape-polymorphic languages such as SaC, we can define inner increments for arrays `a` of arbitrary rank in the same fashion:

```
with {
  ( 0*shape(a)+1 <= iv < shape(a)-1) : a[iv] + 1;
} : modarray( a);
```

The expressive power of with-loops allows them to be used as vehicle to implement all built-in array operations in SaC, including `take`, `drop`, and `++` from above. While this is very convenient from the perspective of language design and implementation, it comes at a price: even trivial operations such as element-wise arithmetics require explicit index-range and shape specifications. Often, this does not matter, as shape-polymorphism enables the abstraction of generic operators such as an element-wise addition for arrays of arbitrary rank. However, in cases where many new operators are needed, the necessity to express shape ranges and ranks adds a lot of verbosity.

Looking at textbooks from Mathematics and Physics, we can see that these have the very same issues when describing tensor calculi. In typical tensor notations, it is customary to omit index ranges whenever they are "obvious" from the context. Likewise, index-ranges of summations and even the summation symbols themselves are omitted based on the occurrence of indices on both sides of equations.

In this paper, we describe a shortcut notation for with-loops and array comprehensions in general which imitates the tensor notations while, at the same time, guaranteeing a non-ambiguous semantics. While the idea of array comprehensions is by no means

new, being able to omit boundaries and shapes becomes rather challenging once shape-polymorphic specifications are involved. Existing approaches are typically rather limited in their applicability due to limitations in the static tractability of these shapes.

The contributions of this paper are threefold:

- we define a new notation for array comprehensions that fully support shape-polymorphic codes; it has the same expressiveness as the with-loops in SaC and can be used as a replacement;
- we provide a formal specification of the translation of these expressions into with-loops; and
- we extend the ability to infer missing shape specifications through a novel rewrite system which builds on the binding scope analysis from [7].

2 WITH-LOOPS

In this section we briefly recap the notion of the With-Loop construct in SaC. The construct can be understood from the following example:

```
with {
   ([2] <= iv < [5]) : iv[0] + 1;  // generator part
} : genarray                       // operator
             ([8],                 // shape of the result
              42)                  // default element
```

which evaluates `[42, 42, 3, 4, 5, 42, 42, 42]` — a 1-dimensional array of 8 elements. Each with-loop consists of one or many generator expressions and an operator. Each generator expression defines a hyperrectangular set of indices and the expression that is evaluated for every index. In the above example, the index set is {[2], [3], [4]} and the expression is `iv[0] + 1`, where `iv` binds to each index from the given index set. Finally, the `genarray` operator constructs a new array. This operator takes two parameters: the shape of the array to be constructed and a default element which is being used for all those element positions that are not covered by any of the generators.

In case of multiple generators with overlapping index sets the value from the first definition is taken. A slight variation of our example:

```
with {
   ([2] <= iv < [5]) : iv[0] + 1;  // generator 1
   ([4] <= iv < [6]) : 0;          // generator 2
} : genarray ([8], 42)             // operator part
```

evaluates to `[42, 42, 3, 4, 5, 0, 42, 42]`.

With-Loops allow for non-scalar body expressions. Such nested array constructions need to be homogeneous, *i.e.*, all body expressions as well as the default element need to be of identical shape. This enables nested With-Loops such as:

```
with {
   ([1] <= jv < [2]) : with {
                          ([2] <= iv < [5]) : iv[0] + 1;
                       } : genarray ([8], 42);
} : genarray ([3], [0,0,0,0,0,0,0,0]);
```

which computes a two-dimensional matrix of overall shape `[3,8]`:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 42 & 42 & 3 & 4 & 5 & 42 & 42 & 42 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

As an alternative to the operator `genarray` the With-Loop also supports an operator `modarray`. It takes an array as parameter which serves as a template for the array to be constructed with respect to both: shape and default element. Assuming that the variable `A` evaluates to a vector of 8 elements `[a,b,c,d,e,f,g,h]`, the With-Loop

```
with {
   ([2] <= iv < [5]) : iv[0] + 1;
} : modarray (A);
```

evaluates to the vector `[a,b,3,4,5,f,g,h]`.

Finally, the with-loop supports a `fold` operator that can be understood from the following example:

```
with {
   ([0] <= iv < [10]) : iv[0]      // generator part
} : fold                           // operator
         (+,                       // binary function
          0)                       // default element
```

It computes the sum of first 9 natural numbers. A more detailled description of with-loops in SaC and their semantics can be found in [5] or on http://www.sac-home.org.

3 ON REDUCING THE VERBOSITY OF WITH-LOOPS

While the standard form of the with-loop as introduced in the previous section is very expressive, at the same time, it is very verbose as well: each generator consists of two bounds, an index variable, two relations and a body expression. The operators all require one (**modarray**-case) or two parameters (**genarray**-case and **fold**-case). None of these components can be omitted, even if they seem "obvious" from the context. Looking at a large body of application code, we can identify four different components of with-loops that in many contexts seem to be very obvious from the context.

3.1 Generator Bounds

Many simple with-loops range over the entire index space of the result shape, i.e., they are simple map-like operations. When specifying shape-polymorphic codes this leads to specifications such as the following with-loop which performs a simple increment on all elements of an array `A`:

```
with {
   (0*shape(A) <= iv < shape(A)) : A[iv] + 1;
} : genarray (shape (A), 0);
```

The need to specify the lower and upper bound of the index range has two drawbacks: they render the code harder to read and more painful to write. To ameliorate that situation to some extent, the current SaC language definition supports syntactic sugar for those two bounds in the form of the dot symbol. Depending on its position, the symbol "." refers to the smallest or biggest legitimate index within the result shape, respectively. Using that shortcut, we can write:

```
with {
   (. <= iv <= .) : A[iv] + 1;
} : genarray (shape (A), 0);
```

While this definitely reduces the specificational burden, readability only partly benefits. The overloading of the symbol "." is far from

ideal. In particular in combination with the ability to choose between "<=" and "<" as relations on both sides of the index variable `iv`, the use of the dot-symbol regularly leads to confusion and to subtle programming errors. Ideally, we would like to get rid of the need to always specify exactly two bounds. Furthermore, the applicability of using the dot-symbol is rather limited. For Fold-With-Loops it cannot be applied at all since there is no explicit shape the bounds could be derived from, and non-scalar with-loops cannot leverage the dot notation either. As an example, consider a with-loop where where the generator ranges over all elements of the first axis only and then operates on the entire hyperplane in the generator body. A slight modification of our increment example exposes this:

```
with {
   ([0] <= iv < take ([1], shape (A)) : A[iv] + 1;
} : genarray ( take ([1], shape (A)), def_value);
```

Here, the use of the dot symbol as shortcut for the bounds is not possible although the index vector ranges over all rows.

3.2 Index Vector Components

Some algorithms are specified in terms of individual axes of arrays rather than all axes as in the first increment example above. Matrix Multiplication is a classical example for such a situation:

```
with {
  ([0,0] <= iv < shape (A))
    : with {
       ([0] <= jv < shape (A)[0])
         : A[[iv[0], jv[0]]] * B[[jv[0], iv[1]]];
      } : fold (+, 0);
} : genarray (shape (A), 0)
```

In those cases, it would be convenient to have scalar indices instead of the index vectors `iv` and `jv`. Again, the current definition of SaC caters for this by means of syntactic flexibility at the index variable specification. Matrix multiply can be written as:

```
with {
  ([0,0] <= [i,j] < shape (A))
    : with {
       ([0] <= [k] < shape (A)[0])
         : A[[i, k]] * B[[k, j]];
      } : fold (+, 0);
} : genarray (shape (A), 0)
```

which is closer to the mathematical specification. However, this pattern match style of introducing scalar index variables is limited to cases where we are dealing with a fixed dimensionality. This precludes from using it in shape-polymorphic operations. In order to cope with that, something analogous to pattern match wild-cards would be needed.

3.3 Default Elements

The flexibility to specify with-loops where the generators traverse through hyper planes rather than individual elements is a very powerful tool for writing shape-polymorphic algorithms that apply some operation relative to one or more specific axes only. The second version of increments above is an example for this. Such with-loops can be applied to arrays of arbitrary shape as long as they have at least one axis. Unfortunately, such functionality usually renders the specification of the default element of **genarray**-operators non-trivial. In the second increment example, we simply used the variable `def_value` in the default element position. However, this variable needs to be defined appropriately. Assuming that we want to make as few assumptions on the shape of the array `A` as possible, we need to define `def_value` as an array of values whose shape is identical to that of `A` without its initial component. In SaC, this can be specified as:

```
def_value = with {
            } : genarray ( drop ([1], shape (A)), 0);
```

This default element has several drawbacks. First of all, it is tedious to specify and secondly, the strict semantics of SaC forces the default element to be computed even if, as in our hyper-plane increments example, it turns out that the default element never is actually being used. It would be ideal if we could avoid an explicit default element specification as often as possible.

One might think that such a default element can always be statically inferred based on the aforementioned homogeneity constraint of SaC: it requires all elements of an array to be of the same type and of the same shape which limits nesting to the homogeneous case. Whenever we can statically infer the shape of the body expression, we can infer a suitable default element as well. In the hyperplane increments example, if the array `A` turns out to be of statically known shape `[c1,c2]` the compiler can statically infer that the shape of `A[iv] + 1` is `[c2]` and a suitable default element could be inserted by the compiler. Unfortunately, in the general case this is not possible. Even if we just consider the case where we statically do not know the rank of `A` finding out that the shape of `A[iv] + 1` actually is identical to `drop ([1], shape (A))` is non-trivial, given that selection and addition are not built-in operators in SaC but user-defined functions which internally use some more restricted built-in operations on scalar values. One might consider that computing the lower bound instance of the body expression could solve the problem, i.e., in our example we could compute the expression

```
shape (A[[0]] + 1)
```

Unfortunately, this only works out for non-empty generators. If we assume the shape of `A` to be `[0,6]`, the expression `shape (A[[0]] + 1)` would yield an out of bound access, while `drop ([1], shape (A))` correctly reduces to `[6]`.

3.4 Result Shapes

Another frequent case for seemingly redundant information arises around the result shape provided as first parameter to the **genarray**-operator. Consider again the trivial increment case. The need to specify that the result shape is identical to that of the array `A` seems notationally excessive. While in this simple case the use of the **modarray**-operator can help, in general, an inference of the result shape is undecidable.

3.5 Set Notation

The dilemma we face here is that on the one hand we cannot guarantee that we can infer all missing parts for arbitrary specifications, and on the other hand we do not want to be forced to have unnecessarily verbose specifications for all cases. The set notation previously introduced to SaC [6] offers one possible way out. It

introduces a much terser notation as syntactic sugar for with-loops where the inference of missing components is straight-forward enough to be automated. In this notation, element-wise increment can be defined as:

```
{ iv -> A[iv] +1 }
```

and matrix multiply becomes:

```
{ [i,j] -> sum ({ [k] -> A[i,k] * B[k,j] }) }
```

In these examples, all missing components such as lower and upper bounds, result shape, and the default element are all inferred from the body expression. While this caters for many frequent cases, it has three main shortcomings. First, it only works for those cases where all these components can be inferred; if any one of them cannot be inferred, a full with-loop has to be used as fallback solution. Secondly, the inference of default elements is solved in an ad-hoc fashion by computing the body expression for the lower bound. The case of an empty shape as explained in section 3.3 is not covered appropriately. Finally, the fact that the set notation is only applicable in the "simple" cases means that both, with-loops and set notation need to co-exist. If it turns out that a case that the programmer considers suitable for the use of set notation in fact is not, the expression needs to be rewritten completely.

The goal of this work is to come up with a single unifying notation which is similar in terseness to the set notation but that is expressive enough to replace with-loops entirely.

4 NEW NOTATION

Our new notation is built on the idea of a set comprehensions, but instead of generating a set, we generate a multi-dimensional array:

```
// Variable    Expression    Generator
{  iv       -> e           | g          ;
   ...                                   }
```

Array comprehensions contain one or more generators, each of which is very similar to their counterparts in genarray with-loops. The term *iv* constitutes a bound variable, *e* is an expression as in the with-loop, and *g* defines index positions at which *e* will be evaluated. The final result is obtained by unifying all the parts of the array comprehension.

The following features set our new array comprehensions apart from previous approaches such as the with-loops or set expressions.

Multiple generators. Array comprehensions contain a list of variable-expression-generator triplets. Each triplet represents a part of the index-space that is being computed. For example:

```
{ iv -> 1 | [0] <= iv < [5]  ;
  iv -> 2 | [5] <= iv < [10] }
```

evaluates a vector of 10 elements, where first five of them are ones and the rest are twos.

Simplified generators. Array comprehensions make it possible to omit specification of the lower bound, upper bound or the entire generator. For example, the following forms of expressions are supported:

```
{ iv -> e | iv < ub  }      // upper bound only
{ iv -> e | lb <= iv }      // lower bound only
{ iv -> e }                 // no generator
```

where *e* is the body of the array comprehension and *lb* and *ub* are expressions for the lower and upper bounds. In all three cases, we attempt to infer the full generator. In the first case this is straightforward: the lower bound is simply `ub*0`. Resolving the second and third case rely on an analysis of *e*: we search for occurrences of the index vector variable `iv` as first argument in selection operations in *e*. For example,

```
{ iv -> a[iv] }
```

is a legal expression, with the index range spanning from `0*shape(a)` to `shape(a)`. As array selections may contain arbitrarily complex expressions, it is not always possible to compute the index bound of *e*. In these cases the inference algorithm produces an error and asks the user to provide additional bound constraints for `iv`.

Index Pattern Matching. Index components can be bound to variables:

```
{ [i,j] -> i+j | [i,j] < [3,3] }
```

Additionally, we support two wild-card pattern: a single dot and three dots. A single dot makes it possible to skip the component of an index vector without binding it to a variable. For example `[i,.,j]` defines a tree-dimensional index space, where the first and the third component of the index vector are bound to *i* and *j* correspondingly. The triple dot can be used once in a pattern, and it skips all the index components except those that are specified by the pattern on the left and on the right. For example: `[i,...]` defines an index space of any rank where the first index component is bound to *i*. The `[i,...,j]` expression also defines an index space of any rank, and it binds the first and the last index components to *i* and *j*. Both dot patterns can be used simultaneously: `[.,i,...,j,.]`.

The last partition. The new notation does neither require a specification of a default element nor a result shape per se. Instead, the shape of the default element is inferred. However, as mentioned before, this cannot be done in all cases. We resolve this problem by introducing the convention that the overall shape of any array comprehension needs to be deducible from the last variable-expression-generator triplet. With this convention, we achieve two goals simultaneously: we can provide missing shape information for the entire expression; and we can provide an explicit default element in cases where the inference is not possible. For example, the notation

```
{ [i,j] -> i+j | [3,3] <= [i,j] ;
  iv    -> 0   | iv < [10,10]   }
```

can be viewed as a with-loop with a shape expression `[10,10]` and a default element `0`. If there was no second partition, an inference of the upper bound would be impossible.

Modarray operators. The **modarray**-variant can easily be modeled in the new notation as well, building on the special role of the last partition. Looking at the previous example, we can specify it to act in the same way as a with-loop with an operator **modarray** `(A)`:

```
{ [i,j] -> i+j | [3,3] <= [i,j] ;
  iv    -> A[iv]                }
```

Fold with loops. The new notation does not provide a direct equivalent for the **fold**-variant either. Instead, generically defined reduction combinators need to be used. For example, matrix multiplication can be expressed by

```
{ [i,j] -> sum({ [k] -> A[i,k]*B[k,j] }) }
```

where `sum` is library function that adds all elements of a given array.

Dots in index patterns. Similarly to the dot syntax in the selection operations, we allow to enrich index patterns with two additional symbols: **.** and **...**. A single dot, similarly to the dot in regular expressions, refers to the position in the index vector for which we do not define a bound variable. Triple dots mean zero or more indices in the index pattern, a '*' pattern in regular expressions. Consider these two examples

```
{ [i,.] -> [1,2,3] + i | [i] < [5]}
{ [.,i] -> [1,2,3] + i | [i] < [5]}
```

The body of the comprehensions generates three-element vectors, and the position on the dot in the index vector defines whether **i** iterates over rows or columns. In the first case we compute a matrix of shape [5, 3], and in the second the shape of the result is [3, 5].

One can think about dot patterns as a way to encode a subsequent transposition of the result. With the triple-dot pattern, we get extra expressivity, as we can capture multiple axes without fixing the number of axes a priori. As an example, consider:

```
{ [i,...] -> A + i | [i] < [5]}
{ [...,i] -> A + i | [i] < [5]}
```

If `A` turns out to be a vector, we obtain exactly the same behaviour as above. However, there is no restriction on the dimensionality of `A` at all. For example, if the shape of `A` turns out to be [2, 2], we get results of the shape [5, 2, 2] and [2, 2, 5], respectively.

Finally, triple dots can be used in the middle of the index pattern as, for example, in:

```
{ [i,...,j] -> [[1,2],[3,4]] + i | [i,j] < [5,5]}
```

which evaluates to an array of shape [5, 2, 2, 5]. To avoid ambiguity, we restrict the use of triple dots to a maximum of one per pattern. Dots and triple dots can be combined in a single pattern without any further restrictions.

5 TRANSLATING TENSOR COMPREHENSIONS

We describe the translation process of the new tensor notation into with-loops, allowing ourselves to omit some technical details for the sake of better readability. The translation happens in three consecutive rewrite steps.

First, we *infer* all missing *upper/lower bounds* for index variables of a given tensor comprehension. We denote this traversal with $[\![\_]\!]_{\text{BI}}$, and an abstract example application looks like this:

$$\{ [\![\mathbf{[i,.,.,j,k,.]} \texttt{ -> } e \texttt{ ; } \gamma]\!]_{\text{BI}} \} = \{ \mathbf{[i,.,.,j,k,.]} \texttt{ -> } e \mid l \texttt{ <= } \mathbf{[i,j,k]} \texttt{ < } u \texttt{ ; } \gamma' \}$$

That is, it computes lower and upper bounds l and u and inserts them as constraints on the index variables **i,j,k**.

Secondly, the rewrite scheme $[\![\_]\!]_{\text{ED}}$ *eliminates all dots in index patterns.* We do so by introducing fresh variables for individual dots, inferring their ranges and merging them into the constraints at their corresponding positions. Finally, we add a selection operation into the body expression. For our running example, we get:

$$\{ [\![\mathbf{[i,.,.,j,k,.]} \texttt{ -> } e \mid l \texttt{ <= } \mathbf{[i,j,k]} \texttt{ < } u \texttt{ ; } \gamma']\!]_{\text{ED}} \}$$
$$= \{ \mathbf{[i,t_1,t_2,j,k,t_3]} \texttt{ -> } e[[\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3]] \mid l' \texttt{ <= } \mathbf{[i,t_1,t_2,j,k,t_3]} \texttt{ < } u' \texttt{ ; } \gamma'' \}$$

In the final rewrite step, $[\![\_]\!]_{\text{W}}$ generates with-loops from tensor comprehensions with explicit lower/upper bounds and without any index wild-cards left:

$$[\![\{ \mathbf{[i,t_1,t_2,j,k,t_3]} \texttt{ -> } e' \mid l' \texttt{ <= } \mathbf{[i,t_1,t_2,j,k,t_3]} \texttt{ < } u' \texttt{ ; } \gamma'' \}]\!]_{\text{W}} =$$

```
with {
  (l' <= [i, t1, t2, j, k, t3] < u'): e'
  ...
} : genarray (...)
```

5.1 Notation

We use semantic brackets to denote the rewrite rules, as our rewrites are compositional. For readability we introduce a few typographical conventions when presenting the rewrites. We use **bold face** for the concrete syntax when constructing or pattern-matching on a SaC expression. We use blue *math* font to denote sub-expressions that we either obtain as arguments or compute during rewrites. Finally, we use green standard font to introduce meta-values or helper meta-functions that operate on SaC expressions.

When we pattern-match on partitions of tensor comprehensions, p ; γ we assume that p is bound to the first partition in the list and γ is the rest of the list, which could be also empty which we denote with $\emptyset$.

5.2 Bound Inference

Bound inference is performed locally for every partition of the tensor comprehension. One extra bit of information that is required for the analysis is whether the partition we are working with is the last one, *i.e.* whether it determines the shape of the overall result. We structure our rewrite in a *map*-like style, and we consider separately four kinds of partitions:

- no bounds i -> e
- lower bound i -> e | l <= $\tilde{i}$
- upper bouns i -> e | $\tilde{i}$ < u
- full i -> e | l <= $\tilde{i}$ < u

No bounds. We start with the case when the partition does not provide bounds. For that case we obtain:

$$[\![\emptyset]\!]_{\text{BI}} = \emptyset$$
$$[\![i \texttt{ -> } e \texttt{ ; } \gamma]\!]_{\text{BI}} = i \texttt{ -> } e \mid l \texttt{ <= } \tilde{i} \texttt{ < } u \texttt{ ; } [\![\gamma]\!]_{\text{BI}}$$

where

$$\tilde{i} = \text{strip-dots } i$$
$$s = \text{find-shapes } \tilde{i}\ e$$
$$\begin{cases} l = u\texttt{*}\mathbf{0} \wedge u = s & \text{is-complete } s \\ l = \texttt{.} \wedge u = \texttt{.} & \neg\text{is-complete } s \wedge \gamma \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

The first rule makes sure that our recursive rewrite terminates when the last partition is done. The second rule inserts the upper

and the lower bounds for the index variables. These are computed from the indices in the index pattern i and the body expression e as follows:

First, we derive from the index pattern i which may contain multiple **.** and a single **...** a version without dots referred to by $\tilde{i}$. We do so formally by means of a helper function strip-dots which eliminates **.** and **...** from the index-pattern, preserving the order of the remaining variables. In case the argument is not an index pattern, it acts as identity function:

strip-dots **[i,.,k,...,m]** = **[i,k,m]**

strip-dots **iv** = **iv**

Then, the find-shapes meta-function traverses its second argument and collects all the information about the ranges for the variables from the first argument. The precision of this analysis impacts on the number of tensor comprehensions that can be handled. The number of methods one can use is very large: data flow, interval analysis, abstract interpretation. In it simplest form, we inspect whether variables are found in selections and per each variable we compute the minimum of upper bounds. For example:

find-shapes **[i,j] (a[j,i]+b[i])**

= **[min(shape(a)[1], shape(b)[0]), shape(a)[0]]**

In case the information is not present, or the analysis is too weak, find-shapes may return a partial result. For example:

find-shapes **[i,j] a[i]** = **[shape(a)[0], _]**

We use is-complete to check that each variable has a shape annotation. Finally, if the shape information is partial, we use **.** as bounds in case the partition is not last or we fail with an error message otherwise. The latter is indicated by the $\bot$ symbol.

Lower Bound. For the case when only a lower bound is specified, we use a similar rule, except that $\tilde{i}$ already has been specified by the user. The appropriate relation between i and $\tilde{i}$ is guaranteed by syntactical well-formedness criteria. We obtain:

$$[\![\ i \text{ -> } e \mid l \text{ <= } \tilde{i}\ ;\ \gamma\ ]\!]_{\mathrm{BI}} = i \text{ -> } e \mid l \text{ <= } \tilde{i} < u\ ;\ [\![\ \gamma\ ]\!]_{\mathrm{BI}}$$

where

$$s = \text{find-shapes } \tilde{i}\ e$$

$$u = \begin{cases} \textbf{take(shape(}l\textbf{), }s\textbf{)} & \text{is-complete } s \wedge \neg\text{pat } i \wedge l \neq \textbf{.} \\ s & \text{is-complete } s \wedge (\text{pat } i \vee l = \textbf{.}) \\ \textbf{.} & \neg\text{is-complete } s \wedge \gamma \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

The main difference in the computation of the upper bound u between this rule and the rule for the no-bounds case above stems from the need to ensure that both, the lower and the upper bound need to be of the same length. While this is inevitable in case we have an index pattern, for an index variable this may not hold; here, an arbitrary prefix of s can be chosen. In the case of no existing bounds we always choose the maximum length. In this case, we need to ensure compliance with the lower bound. We use a predicate pat which is true for index pattern but false for an index variable in order to distinguish those two cases. If we are dealing with an index variable (¬pat), we choose a prefix from s of the same length as the given lower bound l.

Upper Bound. When the upper bound is given but the lower bound is missing, we compute the lower bound by multiplying the upper bound with zero. The only exception is when the last partition has u = **.** — in this case rewrite fails with an error message. We obtain as rule:

$$[\![\ i \text{ -> } e \mid \tilde{i} < u\ ;\ \gamma\ ]\!]_{\mathrm{BI}} = i \text{ -> } e \mid l \text{ <= } \tilde{i} < u\ ;\ [\![\ \gamma\ ]\!]_{\mathrm{BI}}$$

where

$$l = \begin{cases} \bot & \gamma = \emptyset \wedge u = \textbf{.} \\ u\textbf{*0} & \text{otherwise} \end{cases}$$

Full Generator. When both bounds are given, we only have to check that the last partition does not contain **.** as the upper bound.

$$[\![\ i \text{ -> } e \mid l \text{ <= } \tilde{i} < u\ ;\ \gamma\ ]\!]_{\mathrm{BI}} = \alpha\ ;\ [\![\ \gamma\ ]\!]_{\mathrm{BI}}$$

where

$$\alpha = \begin{cases} \bot & \gamma = \emptyset \wedge u = \textbf{.} \\ i \text{ -> } e \mid l \text{ <= } \tilde{i} < u & \text{otherwise} \end{cases}$$

5.3 Eliminating Dots

After the bounds inference has been applied, we have ensured that all partitions have lower and upper bounds, either in terms of concrete expressions or in terms of the dot symbol. Before translating these into with-loops, we want to elide dots in index patterns as these are not admissible in with-loops, *i.e.*, we want to transform partitions of the form

- **[i,.,k]** -> $e \mid l$ <= **[i,k]** < u
- **[i,.,...,k]** -> $e \mid l$ <= **[i,k]** < u

into partitions without dots left of the arrow symbol. Now we consider the two cases (with or without **...**) in separation.

Single dots. We start with the case when the partition we are dealing with pattern-matches on the index variables that do not contain **...** pattern, *i.e.* $i = [j_1, \ldots, j_n] \wedge j_i \in \{\textbf{.}, \mathbf{x}_i\}$. In this case we introduce new variables for each dot pattern, and we select the resulting expression at these index variables. We get:

$$[\![\ i \text{ -> } e \mid l \text{ <= } \tilde{i} < u; \gamma\ ]\!]_{\mathrm{ED}} = \tilde{j} \text{ -> } e\textbf{[[}t\textbf{]]} \mid l' \text{ <= } \tilde{j} < u'\ ;\ [\![\ \gamma\ ]\!]_{\mathrm{ED}}$$

where

$$\begin{aligned} \mathrm{m} &= \text{count-dots } i \\ t &= \mathbf{t}_1, \ldots, \mathbf{t}_{\mathrm{m}} \\ \tilde{j} &= \text{merge } i\ \tilde{i}\ t \\ l' &= \text{merge } i\ l\ \textbf{genarray([m], 0)} \\ u' &= \text{merge } i\ u\ \textbf{take([m], shape(}\bar{e}\textbf{))} \\ \bar{e} &= \text{subst } \tilde{i}\ e\ \textbf{genarray([}\text{count-var } \tilde{i}\textbf{], 0)} \end{aligned}$$

First we compute the number of dots in the pattern and bind it to m. Note that m is a meta-variable, but when we later use it in the object language we implicitly assume that we build an integer term that contains m as a value. After that we introduce m fresh variables, and we compute $\tilde{j}$ by substituting dots in i with the fresh

variables. The merge function works as follows:

$$\text{merge}\ (. :: xs)\ \tilde{i}\ (s :: ss) = s, \text{merge}\ xs\ \tilde{i}\ ss$$
$$\text{merge}\ (\mathbf{v} :: xs)\ (k :: \tilde{i})\ ss = k, \text{merge}\ xs\ \tilde{i}\ ss$$
$$\text{merge}\ xs\ .\ ss = .$$

Finally, as we introduces m fresh variables, the body of the with-loop returns a non-scalar result. In order to compute the shape of this result we take the last m components of the **shape**($\bar{e}$), where $\bar{e}$ is derived from the expression e by substituting all occurrences of index variables by zeros. We defer a discussion of the validity of this choice to the next section as we use this technique for determining the shape of the expressions e in several rules.

Triple Dots. The other case we are dealing with are index patterns which contain '...', *i.e.* $i = [j_1, \ldots, j_n] \wedge j_i \in \{., \mathbf{x}_i, \ldots\}$. As per our syntactical restrictions ... can only occur once, we can split the index pattern into three parts: the left hand side i, the ... and the right hand side j. Both, i and j may be empty. With it, we obtain the following rule:

$$\begin{aligned}[[\ [i\ ++\ \ldots\ ++\ j]\ \text{->}\ e \mid l <= \tilde{i} < u;\ \gamma\ ]]_{\text{ED}} \\ = \tilde{j}\ \text{->}\ e[[t]] \mid l' <= \tilde{j} < u';\ [[\ \gamma\ ]]_{\text{ED}}\end{aligned}$$

where

$$\begin{aligned}\text{m} &= \text{count-dots}\ i \\ \text{n} &= \text{count-dots}\ j \\ t = t_l\ ++\ \mathbf{t}_*\ ++\ t_r &= \mathbf{t}_1, \ldots, \mathbf{t}_\text{m}, \mathbf{t}_*, \mathbf{t}_{\text{m}+2}, \ldots, \mathbf{t}_{\text{m}+1+\text{n}} \\ I &= [i\ ++\ \ldots\ ++\ j] \\ \tilde{j} &= \text{merge-d}\ I\ \tilde{i}\ t_l\ \mathbf{t}_*\ t_r\end{aligned}$$

and

$$\begin{aligned}l' &= \text{merge-d}\ I\ l \\ &\quad \mathbf{genarray}([\text{m}], \mathbf{0}) \\ &\quad \mathbf{genarray}([\mathbf{dim}(\bar{e})\text{-m-n}], \mathbf{0}) \\ &\quad \mathbf{genarray}([\text{n}], \mathbf{0}) \\ u' &= \text{merge-d}\ I\ u \\ &\quad \mathbf{take}([\text{m}], \mathbf{shape}(\bar{e})) \\ &\quad \mathbf{drop}([\text{m}], \mathbf{drop}([\text{-n}], \mathbf{shape}(\bar{e}))) \\ &\quad \mathbf{take}([\text{-n}], \mathbf{shape}(\bar{e})) \\ \bar{e} &= \text{subst}\ \tilde{i}\ e\ \mathbf{genarray}([\text{count-var}\ \tilde{i}], \mathbf{0})\end{aligned}$$

We use exactly the same idea as before, with a minor tweak that we have to introduce fresh variables for the . in i and j and a new variable for ... as well. As before, dots and triple dots get replaced with fresh variables and then we select the body of the generator with these variables. The merge-d function accounts for these three cases as follows:

$$\begin{aligned}&\text{merge-d}\ (i\ ++\ \ldots\ ++\ j)\ (xl\ ++\ xd\ ++\ xr)\ ll\ dd\ rr \\ &\qquad = (\text{merge}\ i\ xl\ ll)\ ++\ dd\ ++\ (\text{merge}\ j\ xr\ rr)\end{aligned}$$

5.4 Tensor to With-loop

After the bounds are inferred and dots are eliminated, tensor notation is just syntactical sugar for with-loops without explicit specifications of result shapes and default elements. We derive these from the last partition leading to an overall rule:

$$[[\ \{\ \tilde{i}_1\ \text{->}\ e_1 \mid l_1 <= \tilde{i}_1 < u_1\ ;\ \ldots\ ;\ \tilde{i}_n\ \text{->}\ e_n \mid l_n <= \tilde{i}_n < u_n\ \}\ ]]_{\text{W}} =$$

```
with {
  (l_1 <= ĩ_1 < u_1): e_1
  ...
  (l_n <= ĩ_n < u_n): e_n
} : genarray (u_n, genarray(shape (ē_n), zero(ē_n)))
```

As before in Section 5.3, $\bar{e}_n$ serves as vehicle to determine the shape of the default element. It is an expression where $\tilde{i}_n$ is substituted by zeroes. That is:

$$\bar{e}_n = \text{subst}\ e_n\ \tilde{i}_n\ \mathbf{genarray(shape(}l_n\mathbf{), zero(}l_n\mathbf{))}$$

6 DEFAULT ELEMENT INFERENCE

As explained in section 3, the inference of default elements boils down to figuring out the shape of the body expression of at least one generator. In particular when writing generic, shape-polymorphic functions this often is statically undecidable. Let us revisit one of the examples from subsection 3.3:

```
with {
   ([0] <= iv < take ([1], shape (A))) : A[iv] + 1;
} : genarray (take ([1], shape (A)), def_value);
```

In a shape-polymorphic context, we have no knowledge about the shape of `A` other than that `A` has to be at least one dimensional. The shape of `default_value` depends on the shape of `A`. As programmers, we may immediately spot that the shape of the default element should be `drop([1],shape(A))`, however, the current system requires this to be stated explicitly.

The formal transformation scheme presented in the previous section takes a very naive approach towards inferring a default element. It generates an expression that computes one of the elements of the array, takes its shape, and computes an array of zeros with that shape as default element. In the example above, this shape computation entails computing the expression `shape (A[[0]] + 1)` which we derived from replacing the index variable with the lower bound. While this may look like a good solution at first glance it has two major drawbacks:

Firstly, the computation of the chosen generator expression is redundant. While the computation of the value itself may be reused for the overall result, the bigger problem lies in the memory allocation that happens whenever the generator expression evaluates to a non-scalar array. At runtime, the default element needs to be computed prior to allocating the memory of the overall array comprehension. This entails, in practice, we almost never can reuse that memory. Instead we typically free it as soon as the shape of that expression has been used or, if we reuse the computed values as well, as soon as all the values have been copied into the overall result array of the with-loop. If the default element is large, the ensuing memory traffic often outweighs a re-computation of the values themselves adding to the overhead incurred.

Secondly, since the generator can be empty, there may be no such instance to compute. To make matters worse, this is a typical scenario when applying shape-polymorphic functions to borderline cases where the axis that the with-loop computes on turns out to be empty. In our given example, consider the case where the shape of the array `A` turns out to be `[0,2048]`. Not only is the generator

empty, but the (strict) computation of `shape (A[[0]] + 1)` leads to an out-of bounds access.

One of the key contributions of this paper is an improvement to the default element problem; we propose a rewrite system for our default-element expressions that

- asserts conformity of all generator body instances;
- reduces the shape inference overhead;
- guarantees that no new out-of-bounds errors are being introduced by the desugaring of tensor comprehensions.

Jointly, this does not only improve the efficiency of the derived code, it also provides a shape homogeneity guarantee for all acceptable tensor comprehensions with inferred default elements.

As mentioned before, all instances of generator expressions need to evaluate to arrays of identical shape. For all practically relevant cases, this implies that the shape of these expressions solely depends on the shapes of all arrays that are referenced in the generator bodies, not their values. The rationale behind this observation is that if the shape depends on the values, chances are very high that the resulting shapes differ between some instances leading to an inhomogeneity error. Hence, the key observation is: if we manage to identify the function bodies that adhere to this "shape homogeneity" and if we manage to construct a function that computes the result shape from the shapes of the relatively free variables in the with-loop bodies, then we can compute the default shapes without computing any generator instances.

The first key idea is to leverage the pre-existing static analysis named "binding scope analysis" [7]. It identifies for arbitrary SaC expressions how much information about the relatively free variables is needed in order to compute the value, shape, or dimensionality of the expression, respectively. We can readily use this inference to identify whether a generator body is shape homogeneous or not. If we apply that analysis to the expression `(A[[0]] + 1)`, we obtain the information that for the shape of this expression it suffices to know the shape of `A`.

The second key idea here is to leverage the binding scope analysis in order to derive a rewrite mechanism which transforms the generator body so that it solely computes the shape of all instances from the shapes of all free variables. For our example, we hope that `shape (A[[0]] + 1)` is rewritten to some code that computes the same as `drop([1],shape(A))`. This would not only reduce the computational overhead but it would also resolve the empty shape problem entirely. This rewrite becomes possible as the binding scope analysis ensures that shape information of the relatively free variables indeed is sufficient to compute the shape of the body expression.

6.1 Formalising the Idea

The binding scope analysis described in [7] is defined in terms of a first order applied lambda calculus which can be seen as a stripdown version of SaC. The key idea of the binding scope analysis is to distinguish between four different levels of information about an array:

- full knowledge of the value; we will denote this with $\mathcal{F}$ in the sequel;
- shape knowledge of the array only; we will denote this with $\mathcal{S}$ in the sequel;
- dimensionality knowledge of the array only; we will denote this with $\mathcal{D}$ in the sequel; or
- no knowledge knowledge about the array at all, which we will denote with $\mathcal{N}$.

The analysis annotates all functions with propagation vectors. They determine which of the four levels ($\mathcal{F}$,$\mathcal{S}$,$\mathcal{D}$,$\mathcal{N}$) are needed in the argument value in order to compute a specifiable level of result. We use the notation $D_{\mathcal{X}}(\ fun\ )$ to denote the level of argument knowledge needed for the given function fun when looking for the level $\mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{D}, \mathcal{N}\}$ of the body expression. For example, we have

$$D_{\mathcal{F}}(\ \lambda\texttt{x.shape(x)}\ ) = \mathcal{S}$$
$$D_{\mathcal{S}}(\ \lambda\texttt{x.shape(x)}\ ) = \mathcal{D}$$
$$D_{\mathcal{F}}(\ \lambda\texttt{x.42}\ ) = \mathcal{N}$$
$$D_{\mathcal{S}}(\ \lambda\texttt{x.take(x,[1,2,3,4,5]}\ ) = \mathcal{F}$$

In order to know the value of applying the shape function, it suffices to know the argument's shape. For finding its shape knowledge the dimensionality of the argument is sufficient. Constant functions such as the third example do not require any information of the argument, and the take function requires full knowledge of its first argument to just figure out the overall shape. For details on how these propagation vectors can be inferred in general, the interested reader may refer to [7]. In the context of this paper, we assume this information to be available.

In the sequel, we provide a formalisation of a rewrite system that rewrites terms of a simple applied λ-calculus λ_{SaC} for whose functions the aforementioned binding scope information is available. While full fledged SaC is richer in terms of syntactic sugar and does not support currying of functions at all, the applied λ-calculus we use here is more amenable for a terse presentation. A projection of the presented transformation to full SaC is straight forward; we present an example to this effect in the next subsection.

The syntax of λ_{SaC} is shown in Fig. 1. We have reduced it to the

$$\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Const} \\
& | & \mathit{Id} \\
& | & \lambda\,\mathrm{Id}\,.\,\mathrm{Expr} \\
& | & \mathit{Prf} \\
& | & (\mathit{Expr}\ \mathit{Expr}) \\
& | & \textbf{with}\ \{ \\
& & \quad (\mathit{Expr} <= \mathit{Id} <= \mathit{Expr}) : \mathit{Expr}; \\
& & \}\ :\ \textbf{genarray}(\mathit{Expr}\,,\ \mathit{Expr}) \\
\\
\mathit{Prf} & \Rightarrow & \textbf{shape} \\
& | & \textbf{dim} \\
& | & \textbf{sel} \\
& | & \ldots
\end{array}$$

Figure 1: The syntax of λ_{SaC}.

absolute minimum: besides constants, variables, abstractions, and applications, we only add the notion of primitive operations as well as the central language construct of SaC, the with-loop. In contrast to SaC, where operations like **shape**, **dim**, and **sel** are user-defined

and much more generic, here, we assume these to be primitives of the language. For example, we assume that **sel** performs scalar selections only. Another simplification from SaC is the restriction to single argument functions as well as the omission of an explicit recursion construct as well as an omission of conditionals. While both of these play a vital role in the inference of the binding scope analysis as explained in [7], for the transformation described here, they do not play a role. Last not least, we assume the absence of higher-order functions since neither SaC nor the binding scope inference support functions that are passed around as arguments.

We describe the transformation of expressions in terms of three translation schemes named $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{D}$. Each of these is used to translate a given expression to an expression that computes the specified level of that expression only. For a given 2 by 3 matrix `[[1,2,3],[4,5,6]]`, we expect $\mathcal{F}$ to act as identity, $\mathcal{S}$ to result in `[2,3]` and $\mathcal{D}$ to deliver the value 2.

We define these translation schemes using rules of the form

$$\mathcal{X}\,[\![\ e\ ]\!]\,\varepsilon\ = e'$$

where $\mathcal{X} \in \{\mathcal{F}, \mathcal{S}, \mathcal{D}\}$ denotes the scheme, e denotes the expression to be rewritten, and ε denotes an environment that for each free variable in e contains a pair $(v, \mathcal{X}')$ indicating to which level the corresponding variable at runtime will have been evaluated. Finally, e' denotes the result of the translation process. Fig. 2 shows the rewrite rules for all three schemes. As we can see from the rules, $\mathcal{F}$ drives the full evaluation into all components of that expression, unless we are dealing with an application of a function that does not require the full values of the arguments. In those cases, the arguments are translated with the corresponding lower level and the body of the function is traversed with an entry in the environment carrying the information to which level the argument has been evaluated. In cases where an argument is not being needed at all (level $\mathcal{N}$), we even eradicate the entire function application. This principle is applied to the primitive functions such as **shape**, **dim**, or **sel** as well. The main difference here is that the body is rewritten immediately and no extension of the environment is needed.

When looking at the rules for $\mathcal{S}$, the first difference to the scheme $\mathcal{F}$ we observe is that variables are now in some cases translated in an application of **shape**. This happens if and only if the variable has been marked as fully evaluated ($\mathcal{F}$) in the environment; otherwise, they are left as is. This stems from the fact that the binding scope analysis provides the maximum level of an argument needed in a function's body. Therefore, it is possible that an argument has been fully evaluated even though, in a given sub-context, only a lower level is needed. At the same time, the analysis guarantees that the annotated level never is lower than the any level needed within a function body. Consequently, we can never encounter a level lower than $\mathcal{S}$ in scheme $\mathcal{S}$. The mechanism for function applications in scheme $\mathcal{S}$ is similar to that of $\mathcal{F}$, it potentially switches the scheme for the arguments and extends the environment when traversing the body. The most interesting rule here is that of the with-loop. Here we see, how the entire body of the with-loop is abandoned as the shape of the result can solely be determined by a concatenation of the shape expression and the shape of the default expression.

The rules for scheme $\mathcal{D}$ follow in the same vein. The main difference here is that applications of the primitive functions as well as the with-loop get even simpler.

$$\mathcal{F}\,[\![\ v\ ]\!]\,\varepsilon = v$$
$$\mathcal{F}\,[\![\ const\ ]\!]\,\varepsilon = const$$
$$\mathcal{F}\,[\![\ (\lambda x.e\ a)\ ]\!]\,\varepsilon = \begin{cases} \mathcal{F}\,[\![\ e\ ]\!]\,\varepsilon & \text{iff } D_{\mathcal{F}}(\ \lambda x.e\ ) = \mathcal{N} \\ (\mathcal{F}\,[\![\ \lambda x.e\ ]\!]\,\varepsilon\ \ \mathcal{X}\,[\![\ a\ ]\!]\,\varepsilon\ ) & \text{iff } D_{\mathcal{F}}(\ \lambda x.e\ ) = \mathcal{X} \in \{\mathcal{F},\mathcal{S},\mathcal{D}\} \end{cases}$$
$$\mathcal{F}\,[\![\ \lambda x.e\ ]\!]\,\varepsilon = \lambda x.\ \mathcal{F}\,[\![\ e\ ]\!]\,\varepsilon' \quad \text{where } \varepsilon' = \varepsilon \oplus (x, D_{\mathcal{F}}(\ \lambda x.e\ ))$$
$$\mathcal{F}\,[\![\ (\mathbf{shape}\ e)\ ]\!]\,\varepsilon = \mathcal{S}\,[\![\ e\ ]\!]\,\varepsilon$$
$$\mathcal{F}\,[\![\ (\mathbf{dim}\ e)\ ]\!]\,\varepsilon = \mathcal{D}\,[\![\ e\ ]\!]\,\varepsilon$$
$$\mathcal{F}\,[\![\ ((\mathbf{sel}\ e_{iv})\ e)\ ]\!]\,\varepsilon\ = ((\mathbf{sel}\ \mathcal{F}\,[\![\ e_{iv}\ ]\!]\,\varepsilon\ )\,\mathcal{F}\,[\![\ e\ ]\!]\,\varepsilon\ )$$
$$\mathcal{F}\left[\!\!\left[\begin{array}{l}\mathbf{with}\ \{\\ \quad (e_l <= v <= e_u) : e_b;\\ \}\ :\ \mathbf{genarray}(e_s, e_d)\end{array}\right]\!\!\right]\varepsilon$$
$$= \begin{array}{l}\mathbf{with}\ \{\\ \quad (\mathcal{F}\,[\![\ e_l\ ]\!]\,\varepsilon\ <= v <= \mathcal{F}\,[\![\ e_u\ ]\!]\,\varepsilon\ ) : \mathcal{F}\,[\![\ e_b\ ]\!]\,\varepsilon\ ;\\ \}\ :\ \mathbf{genarray}(\mathcal{F}\,[\![\ e_s\ ]\!]\,\varepsilon\ ,\ \mathcal{F}\,[\![\ e_d\ ]\!]\,\varepsilon\ )\end{array}$$

$$\mathcal{S}\,[\![\ v\ ]\!]\,\varepsilon = \begin{cases} \texttt{shape}\ (v) & \varepsilon(v) = \mathcal{F} \\ v & \varepsilon(v) = \mathcal{S} \end{cases}$$
$$\mathcal{S}\,[\![\ const\ ]\!]\,\varepsilon = shape(const)$$
$$\mathcal{S}\,[\![\ (\lambda x.e\ a)\ ]\!]\,\varepsilon = \begin{cases} \mathcal{S}\,[\![\ e\ ]\!]\,\varepsilon & \text{iff } D_{\mathcal{S}}(\ \lambda x.e\ ) = \mathcal{N} \\ (\mathcal{S}\,[\![\ \lambda x.e\ ]\!]\,\varepsilon\ \ \mathcal{X}\,[\![\ a\ ]\!]\,\varepsilon\ ) & \text{iff } D_{\mathcal{S}}(\ \lambda x.e\ ) = \mathcal{X} \in \{\mathcal{F},\mathcal{S},\mathcal{D}\} \end{cases}$$
$$\mathcal{S}\,[\![\ \lambda x.e\ ]\!]\,\varepsilon = \lambda x.\ \mathcal{S}\,[\![\ e\ ]\!]\,\varepsilon' \quad \text{where } \varepsilon' = \varepsilon \oplus (x, D_{\mathcal{S}}(\ \lambda x.e\ ))$$
$$\mathcal{S}\,[\![\ (\mathbf{shape}\ e)\ ]\!]\,\varepsilon = [\mathcal{D}\,[\![\ e\ ]\!]\,\varepsilon\ ]$$
$$\mathcal{S}\,[\![\ (\mathbf{dim}\ e)\ ]\!]\,\varepsilon = []$$
$$\mathcal{S}\,[\![\ ((\mathbf{sel}\ e_{iv})\ e)\ ]\!]\,\varepsilon = []$$
$$\mathcal{S}\left[\!\!\left[\begin{array}{l}\mathbf{with}\ \{\\ \quad (e_l <= v <= e_u) : e_b;\\ \}\ :\ \mathbf{genarray}(e_s, e_d)\end{array}\right]\!\!\right]\varepsilon\ = (\mathcal{F}\,[\![\ e_s\ ]\!]\,\varepsilon\ \texttt{++}\,\mathcal{S}\,[\![\ e_d\ ]\!]\,\varepsilon\ )$$

$$\mathcal{D}\,[\![\ v\ ]\!]\,\varepsilon = \begin{cases} \texttt{dim}\ (v) & \varepsilon(v) = \mathcal{F} \\ \texttt{shape}\ (v)\texttt{[0]} & \varepsilon(v) = \mathcal{S} \\ v & \varepsilon(v) = \mathcal{D} \end{cases}$$
$$\mathcal{D}\,[\![\ const\ ]\!]\,\varepsilon = dim(const)$$
$$\mathcal{D}\,[\![\ (\lambda x.e\ a)\ ]\!]\,\varepsilon = \begin{cases} \mathcal{D}\,[\![\ e\ ]\!]\,\varepsilon & \text{iff } D_{\mathcal{D}}(\ \lambda x.e\ ) = \mathcal{N} \\ (\mathcal{D}\,[\![\ \lambda x.e\ ]\!]\,\varepsilon\ \ \mathcal{X}\,[\![\ a\ ]\!]\,\varepsilon\ ) & \text{iff } D_{\mathcal{D}}(\ \lambda x.e\ ) = \mathcal{X} \in \{\mathcal{F},\mathcal{S},\mathcal{D}\} \end{cases}$$
$$\mathcal{D}\,[\![\ \lambda x.e\ ]\!]\,\varepsilon = \lambda x.\ \mathcal{D}\,[\![\ e\ ]\!]\,\varepsilon' \quad \text{where } \varepsilon' = \varepsilon \oplus (x, D_{\mathcal{D}}(\ \lambda x.e\ ))$$
$$\mathcal{D}\,[\![\ (\mathbf{shape}\ e)\ ]\!]\,\varepsilon = 1$$
$$\mathcal{D}\,[\![\ (\mathbf{dim}\ e)\ ]\!]\,\varepsilon = 0$$
$$\mathcal{D}\,[\![\ ((\mathbf{sel}\ e_{iv})\ e)\ ]\!]\,\varepsilon = 0$$
$$\mathcal{D}\left[\!\!\left[\begin{array}{l}\mathbf{with}\ \{\\ \quad (e_l <= v <= e_u) : e_b;\\ \}\ :\ \mathbf{genarray}(e_s, e_d)\end{array}\right]\!\!\right]\varepsilon$$
$$= (\ \texttt{sel([0],}\ \mathcal{S}\,[\![\ e_s\ ]\!]\,\varepsilon\ ) + \mathcal{D}\,[\![\ e_d\ ]\!]\,\varepsilon\ )$$

Figure 2: Rewrite rules for the expressions of λ_{SaC} which reduces the computation to the demand determined by the binding scope analysis from [7].

When applying these ideas to fully fledged SaC, the main difference is that we have to convert between sequences of assignments and nested applications of abstractions (let bindings) on the fly and that we have to consider memoisation of the $\mathcal{F}$, $\mathcal{S}$, and $\mathcal{D}$ versions of the top-level,potentially recursive functions in SaC.

6.2 An Example Transformation

We illustrate the effect of the proposed transformation on SaC code by means of the example from Section 3.3:

```
with {
   ([0] <= iv < take ([1], shape (A))) : A[iv] + 1;
} : genarray (take ([1], shape (A)), def_value);
```

Applying our idea, we want to rewrite the body expression `A[iv] + 1` after substituting the index variable `iv` by the lower bound `[0]` into the shape-demand form. As our formalism expects the environment to contain the evaluation status of all free variables, in the example only `A`, we need to initialise this environment. Given that all free variables of the with-Loop body are fully evaluated prior to the with-Loop computation itself, we insert them marked as fully evaluated. Now, we can compute

$$\mathcal{S}\,[\![\ \texttt{A[[0]] + 1}\ ]\!]\,(A, \mathcal{F})$$

We are dealing with an application of the function "+" here which takes two arguments. From the application rule for $\mathcal{S}$, we first have to derive the shape demands for these two arguments.

When looking at the corresponding definition of "+" from the standard library of SaC, we find

```
int[+] +( int[+] A, int B)
{
  shp = _shape_A_(A);
  res = with {
          (. <= iv <= .) : _add_SxS_(_sel_VxA_(iv,A),B);
        } : genarray(shp, 0);
  return( res);
}
```

Note here, that the built-in operations in SaC are prefixed with the symbol "_". They also carry postfixes which indicate restrictions of the admissible argument shapes; "S" refers to scalars, "V" refers to vectors, and "A" denotes arbitrary arrays. The SaC primitive `_sel_VxA_` indeed is identical to the operation **sel** in λ_{SaC}. With this information, we can now come back to the transformation of the addition `A[[0]] + 1` into its shape form. The binding scope analysis of the function + yields that the demands for the shape of the result are $\mathcal{S}$ for the first argument (A) and $\mathcal{N}$ for the second (B). Consequently, our rules elide the second argument and transform both, the function "+" and the expression `A[[0]]` into the shape form; we obtain:

$$(\mathcal{S}\,[\![\ +\ ]\!]\,\varepsilon \quad \mathcal{S}\,[\![\ A[[0]]\ ]\!]\,(A, \mathcal{F})\ )$$

First, let us consider the transformation of "+" into the shape version. When entering the body, we need to add the non-$\mathcal{N}$ parameters into the environment with their corresponding demand. With ε=(A,$\mathcal{S}$), we obtain

$$\mathcal{S}\,[\![\ +\ ]\!]\,\emptyset$$

$$= \mathcal{S}\ \left[\!\!\left[\ \begin{array}{l}(\lambda\ \text{shp}\ .\ \textbf{with}\ \{\\ \qquad (.\ <=\ \text{iv}\ <=\ .)\\ \qquad :\ \text{\_add\_SxS\_(\_sel\_VxA\_(iv,A),B);}\\ \quad \}\ :\ \textbf{genarray}(\text{shp},\ 0);\\ \text{\_shape\_A\_(A)}\ )\end{array}\ \right]\!\!\right]\ \varepsilon$$

Since the binding scope analysis identifies that `shp` is needed as full value in order to compute the shape of the with-loop in the body of the abstraction, we further obtain with $\varepsilon' = \varepsilon \oplus (\text{shp}, \mathcal{F})$:

$$= (\ \mathcal{S}\ \left[\!\!\left[\ \begin{array}{l}\lambda\ \text{shp}\ .\ \textbf{with}\ \{\\ \qquad (.\ <=\ \text{iv}\ <=\ .)\\ \qquad :\ \text{\_add\_SxS\_(\_sel\_VxA\_(iv,A),B);}\\ \quad \}\ :\ \textbf{genarray}(\text{shp},\ 0);\end{array}\ \right]\!\!\right]\ \varepsilon$$
$$\mathcal{F}\,[\![\ \text{\_shape\_A\_(A)}\ ]\!]\,\varepsilon\ )$$

$$= (\ \lambda\ \text{shp}\ .\ \mathcal{S}\ \left[\!\!\left[\ \begin{array}{l}\textbf{with}\ \{\\ \qquad (.\ <=\ \text{iv}\ <=\ .)\\ \qquad :\ \text{\_add\_SxS\_(\_sel\_VxA\_(iv,A),B);}\\ \quad \}\ :\ \textbf{genarray}(\text{shp},\ 0);\end{array}\ \right]\!\!\right]\ \varepsilon'$$
$$\mathcal{F}\,[\![\ \text{\_shape\_A\_(A)}\ ]\!]\,\varepsilon\ )$$

$$= (\ \lambda\ \text{shp}\ .\ (\mathcal{F}\,[\![\ \text{shp}\ ]\!]\,\varepsilon' \ \texttt{++}\ \mathcal{S}\,[\![\ \texttt{0}\ ]\!]\,\varepsilon')\quad \mathcal{F}\,[\![\ \text{\_shape\_A\_(A)}\ ]\!]\,\varepsilon\ )$$
$$= (\ \lambda\ \text{shp}\ .\ (\text{shp}\ \texttt{++}\ \texttt{[]})\quad \mathcal{F}\,[\![\ \text{\_shape\_A\_(A)}\ ]\!]\,\varepsilon\ )$$
$$= (\ \lambda\ \text{shp}\ .\ (\text{shp}\ \texttt{++}\ \texttt{[]})\quad \texttt{A}\ )$$

This leads to a shape variant of "+" of the form:

```
int[.] +_s( int[.] A)
{
  shp = A;
  return shp ++ [];
}
```

Similarly, we compute a shape variant for the selection operation which is generically defined as:

```
double[*] sel( int[.] idx, double[*] array)
{
  new_shape = _drop_SxV_( _sel_VxA_( [0], _shape_A_(idx)),
                          _shape_A_(array));
  res = with {
          ( . <= iv <= . ) {
            new_idx = _cat_VxV_( idx, iv);
          } : _sel_VxA_(new_idx, array);
        } : genarray( new_shape, 0.0);
  return( res);
}
```

For this function, we obtain:

```
int[.] sel_s( int[.] idx, int[.] A)
{
  new_shape = _drop_SxV_( _sel_VxA_( [0], idx), A);
  return new_shape ++ [] ;
}
```

Overall, we get for our default expression:

```
+_s ( sel_s ( [1], _shape_A_(A)))
```

which evaluates to the same value as `drop([1],shape(A))` does.

7 RELATED WORK

There exists a number of syntactic sugars for frequently used data structures. One of the earliest programming languages that added comprehensions as a part of its syntax was SETL [12]. As sets were fist-class objects in SETL, a set comprehension primitive was used to for new sets:

$$\{\ \mathbf{e}(\mathbf{x}_1, \ldots, \mathbf{x}_n)\ ,\ \mathbf{x}_1 \in e_1, \ldots, \mathbf{x}_n \in e_n \mid \mathbf{C}(\mathbf{x}_1, \ldots, \mathbf{x}_n)\ \}$$

the main expression e with n free variables each of which iterates over the corresponding sets, and the entire expression is guarded by the constraint C.

List comprehensions as we know them in Haskell and many other languages for the first time were introduced in KRC by the name ZF expressions [13]. The structure of the comprehension is

very similar to the one from SETL, except it produces the list and free variables iterate over lists as well. For example:

```
[(a,b,c) | a <- [1..30], b <- [1..30], c <- [1..30]
         , a*a+b*b==c*c]
```

generates a list of Pythagorean triplets where each component is less than 30. List comprehensions become very powerful when used in recursive definitions. For example, consider a classical example of the list of prime numbers:

```
si (x:xs) = x : si [n | n<-xs, n `mod` x /= 0]
primes = si [2..]
```

Such style list comprehensions can be found in many other programming languages: Python, R, Julia, F#, *etc.*

Despite having a lot of expressive power, list comprehension gets quite cumbersome if we try to use it for multi-dimensional arrays. For example, if we represent arrays as nested lists, the matrix transpose based on list comprehensions would look like:

```
iota n = [0..n-1]
trans a = [[ x !! i | x <- a] | i <- iota $ length $ a !! 0]
```

We can make this function more readable in many different ways, but unfortunately list comprehensions would not help. The main difficulty is the necessity to maintain the nested structure, *i.e.* being aware at which nesting level we are.

The handle on explicit indexing in the proposed array comprehension makes it possible to solve the above problem as well as obtain rank polymorphism. In the earlier work on axis control notation [6] in SaC a very similar idea has been explored already — a simplified array comprehension with implicit bounds. However, there are two important differences. Firstly, the notation could not entirely replace genarray/modarray with-loops, as it didn't allow to specify explicit bounds. Secondly, and more importantly, the notation was only applicable for the contexts where all the array shapes were known statically. Such a knowledge entirely avoids the problem of the default element shape, as this information is encoded in array types. The current work can be seen as a generalisation of the axis control notation.

A number of attempts have been made to bring the Einstein tensor notation [16] into a programming language. The main point of this notation is that we introduce upper and lower indices, and when the same index variable appears on different levels in the same term, we implicitly assume summation of this term over all the legal values of that index. For example, matrix multiplication would be written as:

$$\sum_{k=1}^{n} A_{ik} B_{kj} = AB = A^i_k B^k_j$$

Note that the notation leaves the upper and lower bounds of all indices completely implicit.

An example of such a system is Tensor Comprehensions [14] by Facebook. This is a framework that has been created to accelerate machine learning operations and it uses the Einstein convention. However, syntactically lower- and higher-level indices are indistinguishable, so the terms look like assignments where the left hand side is an indexed tensor, and the right hand side is an expression. Depending on the kind of assignment being used, the subterms will or will not be summed-up. For example, the matrix multiply is denoted as:

```
def mm(float(N,M) A, float(M,K) B) -> (C) {
  C(i,j)  = 0
  C(i,j) += A(i,k) * B(k,j)
}
```

In [8] the authors also build on the idea that all the tensor operations can be codified as assignments of indexed expressions to an indexed variable. The notation gets quite involved: it introduces multiple classes of indices, operations on them, rules how to identify when the summation happens. The indices can be nested, Kronecker deltas are natively supported, and sub-arrays can be selected. However, to our knowledge the notation only exists as a theoretical framework, and was never picked up by a programming language.

Matlab includes a tensor notation framework as well. In [1] the authors argue that "Notation for tensor multiplication is very complex" and they introduce a number of special cases for matrix-tensor operations only, *e.g.* matrix-tensor multiplication, tensor-vector multiplication, *etc.* Other operations can be implemented by converting tensors into multi-dimensional arrays, defining the operation on arrays and, subsequently, converting the results back into tensors.

All the three above-mentioned frameworks lack the ability to support rank-polymorphic operations, as the classical tensor notation never abstracts over the index vectors. In a way, in relation to with-loops, it forces one to always use pattern matching index-vectors in the with-loops.

The AlphaZ [15] system is a set of tools for program analysis, transformation and parallelisation in the Polyhedral Model [4]. The notation used for program inputs comes very close to what we use for tensor comprehensions, except that the index-spaces can be of arbitrary polyhedral shape. For example, in AlphaZ, we can define a rectangular array **A** as **float A {i,j | 1 <= i < N && 1 <= j < i}**. The computation may be defined in multiple partitions as well, and the expressions are allowed to have self references. For example: **A[i,j] = case {|j==1}: 0; {|i>1&&j>1}:A[i-1,j-1]+1 esac** consists of two partitions and the second one makes a recursive reference to the structure we are defining. At the moment, SaC does neither support non-rectangular index spaces nor recursive tensor comprehensions. However, AlphaZ does not allow to omit any index ranges, and it relies on the polyhedral model to check the validity of the expression.

A number of systems use the notion of iterators to implement list or array style comprehensions [11]. Iterators followed by a typical higher-order combinators like map, reduce or filter can be quite expressive. For example in Rust [10], a list of even squares can be expressed as:

```
(0..10).filter(|x| x % 2 == 0)
       .map(|x| x * x).collect::<Vec<_>>()
```

However, in case of multi-dimensional arrays, an iterator defines the traversal order of the array's index space; parallel executions of such code can be very challenging as demonstrated in [2, 3, 9]. In contrast, with-loops and consequently the new tensor notation are fully data-parallel — each element in the result can be computed independently of others.

8 CONCLUSIONS

This paper introduces an elegant notation for array comprehensions. On the one hand, the notation can be seen as a set comprehension for multi-dimensional arrays. On the other hand, it allows the omission of generators for indices in many cases very similar to the Einstein convention when writing tensor operations. At the same time, the notation makes it possible to express rank-polymorphic comprehensions — something that typical tensor notations do not support.

The proposed notation has been implemented in the context of the array programming language SaC. It is added to the language as a syntactic sugar, which means that all the existing optimisations become immediately applicable to the programs written in that notation. Also, this suggests that the proposed notation can be implemented in other contexts, as we only rely on the existence of with-loops and a certain level of program analysability.

The key challenges of this work have to do with inferring implicit bounds of the generator expressions; and inferring the shape of the default element when transforming tensor comprehensions. The former is a data-flow analysis problem where for each index we track whether that index participates in selections. If it does, we compute the minimal upper bound for that index without incurring out-of-bound errors.

The default element shape inference turns out to be a quite complex task. A tensor comprehension is structured as a sequence of pairs where the first component is an index space and the second one is an expression that is computed in that space. Per SaC semantics, the shape of all the expressions must be the same, so in principle, computing the shape of any of these expression delivers the right answer. However, if the chosen expression is guarded by an empty index space, there is no valid index that we can use during the evaluation. Moreover, when the overall result is an empty array, we are free to "invent" an arbitrary shape. For example, let **a** be an empty array of shape **[0,5]**, in this case the shape of **{ [i] -> reverse(a[i]) }** can be an empty array of any shape, as this inner reverse would be never computed. At the same time, it is natural to assume that the shape of the inner expression should be **[5]**. We propose a technique that makes this idea precise. By relaxing the semantics of our shape computations, we obtain the desired behaviour. As an added benefit, our technique ensures that the shape of that expression solely depends on the shape of the index vector and not its values. Consequently, it provides a homogeneity guarantee which could be leveraged to avoid homogeneity checks at runtime.

Alternatives to the proposed solution always seem to come with restrictions. For example, we could require a programmer to define a precise shape for the entire expression, either via the type system or additional syntax, but our initial goal was to avoid this in the first place. We could restrict all the with-loop bodies to evaluate to scalars, but that would harm expressiveness. Whether there is an alternative approach to solve this problem maintaining expressiveness is a topic of further investigation.

The proposed approach of shape-aware computations based on the demand analysis of programs opens up interesting applications outside the default element problem. It could possibly be used as the base for a new semantic definition of the entire language. If so, it would statically elide computations that do not contribute to the result, similar as normal order or lazy evaluation does. However, it would do so *without* requiring any support for non-strict computations at runtime. The scope of the application of this technique as well as its generalisation are future work.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their many constructive suggestions. This work is supported by the Engineering and Physical Sciences Research Council through grants EP/L00058X/1 and EP/N028201/1.

REFERENCES

[1] Brett W. Bader and Tamara G. Kolda. 2006. Algorithm 862: MATLAB Tensor Classes for Fast Algorithm Prototyping. *ACM Trans. Math. Softw.* 32, 4 (Dec. 2006), 635–653. https://doi.org/10.1145/1186785.1186794

[2] Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. 2015. Parameterized Diamond Tiling for Stencil Computations with Chapel Parallel Iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/2751205.2751226

[3] Bradford L. Chamberlain and Sung-Eun Choi. 2011. User-Defined Parallel Zippered Iterators in Chapel. In *PGAS 2011: Fifth Conference on Partitioned Global Address Space Programming Models*.

[4] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (01 Feb 1991), 23–53. https://doi.org/10.1007/BF01407931

[5] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC - A Functional Array Language for Efficient Multi-threaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427. https://doi.org/10.1007/s10766-006-0018-x

[6] Clemens Grelck and Sven-Bodo Scholz. 2003. Axis Control in Sac. In *Implementation of Functional Languages, 14th International Workshop (IFL'02), Madrid, Spain, Revised Selected Papers (Lecture Notes in Computer Science)*, Ricardo Peña and Thomas Arts (Eds.), Vol. 2670. Springer, 182–198. https://doi.org/10.1.1.540.8938

[7] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. 2006. A Binding Scope Analysis for Generic Programs on Arrays. In *Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland, Revised Selected Papers (Lecture Notes in Computer Science)*, Andrew Butterfield (Ed.), Vol. 4015. Springer, 212–230. https://doi.org/10.1007/11964681_13

[8] Richard A Harshman. 2001. An index formalism that generalizes the capabilities of matrix notation and algebra to n-way arrays. *Journal of chemometrics* 15, 9 (2001), 689–714.

[9] M. Joyner, B. L. Chamberlain, and S. J. Deitz. 2006. Iterators in Chapel. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. 8 pp.–. https://doi.org/10.1109/IPDPS.2006.1639499

[10] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

[11] Lionel Morel. 2002. Efficient compilation of array iterators for Lustre. *Electronic Notes in Theoretical Computer Science* 65, 5 (2002), 19 – 26. https://doi.org/10.1016/S1571-0661(05)80437-2 SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of ETAPS 2002).

[12] J T Schwartz. 1975. Programming an interim report on the SETL project. Part I: generalities. Part II: the SETL language and examples of its use. (6 1975). https://doi.org/10.2172/7196480

[13] D. A. Turner. 2016. *Recursion Equations as a Programming Language*. Springer International Publishing, Cham, 459–478. https://doi.org/10.1007/978-3-319-30936-1_24

[14] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[15] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. *AlphaZ: A System for Design Space Exploration in the Polyhedral Model*. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/978-3-642-37658-0_2

[16] K. Åhlander. 2002. Einstein summation for multidimensional arrays. *Computers & Mathematics with Applications* 44, 8 (2002), 1007 – 1017. https://doi.org/10.1016/S0898-1221(02)00210-9