A Binding Scope Analysis for Generic Programs on Arrays

Clemens Grelck[1], Sven-Bodo Scholz[2], and Alex Shafarenko[2]

[1] Inst. of Software Technology and Programming Languages, University of Lübeck, Germany
Grelck@isp.uni-luebeck.de
[2] Dept of Computer Science, University of Hertfordshire, United Kingdom
{S.Scholz,A.Shafarenko}@herts.ac.uk

Abstract. Performance of generic array programs crucially relies on program specialisation wrt. shape information. Traditionally, this is done in a rather ad hoc fashion by propagating all shape information that is available. When striving for a compositional programming style that adheres to good software engineering principles this approach turns out to be insufficient. Instead, static value information needs to be propagated as well which introduces all the well known problems of partial evaluation in general.

In this paper, we propose a static analysis that identifies to what extent specialisation needs to be employed in order to achieve a certain level of shape information. This narrows the scope of specialisation far enough to make specialisation for shape information feasible despite a compositional programming style. Some examples to this effect are presented.

1 Introduction

Compiling abstract high-level specifications into efficiently executable code is well-known to be a challenging task. Usually, a whole set of complementing optimisations need to be orchestrated properly in order to achieve excellent runtime performance. In the area of array programming, the effectiveness of many optimisations relies on static knowledge of array rank (dimension) and array shape (extent wrt. individual axes). Not only does static knowledge of shapes facilitate many loop related optimisations, it is also essential for eliminating intermediate arrays[LLS98, Sch03] as well as compiler-introduced memory reuse [Can89, GT04].

For most applications in array programming, the majority of array operations are such that the shape of the result can be computed from the shapes, rather than full values, of the arguments. Such operations often are referred to as *uniform operations* [Hui95]. Uniformity enables a straight-forward approach to an effective utilisation of static shape information: Whenever a shape information is statically available it is propagated into all existing function calls by specialising these according to the given shapes. Since most array programs operate on a

A. Butterfield, C. Grelck, and F. Huch (Eds.): IFL 2005, LNCS 4015, pp. 212–230, 2006.

small set of different shapes only, non-termination of specialisation in practice is rarely hit or otherwise can be detected by a compiler fairly easily [Kre03]. For these reasons, array languages such as Fish [JMB98, JS98] or SaC [Sch03] follow that approach.

Unfortunately, uniformity is at odds with a compositional programming style. In contrast to Fish, SaC allows the programmer to successively break down complex (and usually uniform) array operations into compositions of small, rather generic operations similar to those available in Apl. These small array operators typically separate concerns such as inspecting structural properties, selecting parts of an array, or combining arrays into new ones. Unfortunately, the separation of concerns in most cases makes these small operators non-uniform, i.e., the shapes of their results depend on argument values rather than argument shapes only. Typical examples are operations such as `take` or `drop`. These operations select parts of an array by taking or dropping a certain amount of elements, respectively. The number of elements to be dropped or taken is specified as an explicit parameter of these operations which renders the shape of the result dependent on that parameter's value.

Although such a programming style is desirable from a software engineering perspective, it has a strong impact on the performance of such specifications. A specialisation strategy as described above, i.e., based on specialisations to shapes only, leads to a loss of shape information whenever non-uniform operations such as `take` or `drop` are used. As shown in [Kre03], the loss of static shape information can have a significant effect on the overall performance.

One alternative to avoid this potential source of performance degradation would be to specialise functions to argument values whenever these are statically available. However, this would in fact fully embrace the online approach to partial evaluation and, with it, its well-known difficulties: recursive functions introduce undecidability, and the resulting code expansion may outweigh the potential gain in performance (for surveys see [JGS93, Jon96]).

In order to avoid these difficulties, we propose a static program analysis that for each function of a given program infers what level of argument specialisation is required in order to compute the shape of the result. With this information, we can restrict specialisation to argument values to those situations, where this information is crucial for shape inference. In all other situations, a less aggressive specialisation scheme, e.g. specialisation to argument shapes, can be applied. Since Apl-style program compositions usually contain only a small percentage of non-uniform operations it turns out that, by and large, only a few specialisations to argument values are required in order to statically infer all shapes within a large application program.

More generally, the proposed analysis can serve as a "specialisation oracle" that guides the entire specialisation process as the inference algorithm does not only compute the requirements for static shape knowledge, but it also determines the requirements for other levels of static shape information such as static rank knowledge. This additional information can be used for adjusting the specialisation oracle so that it can predict the minimum level of specialisation that is

required for a predefined level of overall shape information. Once the scope of the specialisation has been determined, an online approach towards specialisation suffices for specialising most programs to the predefined level irrespective of whether they have been written in a compositional style or not.

The inference algorithm is described in terms of a subset of SaC [Sch03], which has been adjusted to a fairly generic λ-calculus syntax. This measure allows us to concentrate on the language essentials and it may facilitate transferability of results to other languages. Besides a formal description of the inference, its effectiveness is demonstrated by means of several examples.

The paper is organised as follows: the next section introduces a stripped-down version of SaC, called SaC$_\lambda$. Section 3 discusses the issues of compositional programming and function specialisation by means of a few examples. The main idea of the analysis is presented in Section 4, before Section 5 and Section 6 provide the formal details of it. In Section 7 the formalism is applied to the examples of Section 3. Section 8 relates the work to other approaches towards the specialisation og generic program specifications before some conclusions are drawn in Section 9.

2 SaC$_\lambda$

This paper is based on a stripped-down version of SaC. It contains only the bare essentials of the language and its syntax has been adjusted to a λ-calculus style in order to facilitate transferability of results.

Fig. 1 shows the syntax of SaC$_\lambda$. A program consists of a set of mutually recursive function definitions and a designated main expression. Essentially, ex-

$$
\begin{array}{lll}
\mathit{Program} & \Rightarrow & \left[\, \mathit{FunId} \;=\; \lambda\, \mathit{Id} \left[\; , \mathit{Id} \,\right]^{*} \,.\, \mathit{Expr} \;;\; \right]^{*} \\
 & & \quad \mathbf{main} \;=\; \mathit{Expr} \;; \\
 & & \\
\mathit{Expr} & \Rightarrow & \mathit{Const} \\
 & \mid & \mathit{Id} \\
 & \mid & \mathit{FunId} \;(\; \left[\, \mathit{Expr} \left[\; , \mathit{Expr} \,\right]^{*} \,\right] \;) \\
 & \mid & \mathit{Prf} \;(\; \left[\, \mathit{Expr} \left[\; , \mathit{Expr} \,\right]^{*} \,\right] \;) \\
 & \mid & \mathbf{if}\ \mathit{Expr}\ \mathbf{then}\ \mathit{Expr}\ \mathbf{else}\ \mathit{Expr} \\
 & \mid & \mathbf{let}\ \mathit{Id} = \mathit{Expr}\ \mathbf{in}\ \mathit{Expr} \\
 & \mid & \mathbf{with}(\ \mathit{Expr} <= \mathit{Id} < \mathit{Expr}\ )\ :\ \mathit{Expr} \\
 & & \quad \mathbf{genarray}(\ \mathit{Expr}\ ,\ \mathit{Expr}\ ) \\
 & & \\
\mathit{Prf} & \Rightarrow & \mathbf{shape} \\
 & \mid & \mathbf{dim} \\
 & \mid & \mathbf{sel} \\
 & \mid & * \\
 & \mid & \ldots
\end{array}
$$

Fig. 1. The syntax of SaC$_\lambda$

pressions are either constants, variables or function applications. As SaC does neither support higher-order functions nor name-less functions, abstractions occur at top-level only. Function applications are written in C-style, i.e., with parenthesis around arguments rather than entire applications. It should be noted here that all constants are in fact arrays. Therefore, we use (nestings of) vectors in square-brackets alongside with scalars as notation for constants. SaC_λ provides a few built-in array operators, referred to as primitive functions. Among these are `shape` and `dim` for computing an array's shape and dimensionality (rank), respectively. Furthermore, a selection operation `sel` is provided which takes two arguments: an index vector that indicates the element to be selected and an array to select from. These very basic array operations are complemented by element-wise extensions of arithmetic and relational operations such as `*` and >=, respectively. For improved readability, we use the latter in infix notation throughout our examples.

On top of this language kernel, SaC provides a special language construct for defining array operations in a generic way which is called WITH-loop. For the purpose of this paper, it suffices to consider a restricted form of WITH-loop only. Fully-fledged WITH-loops are described elsewhere, e.g. in [Sch03]. They provide several extensions which primarily relate to programming convenience. Since these extensions do not affect the analysis in principle but would substantially blow up the formal apparatus, we refrain from the fully-fledged version.

As can be seen from Fig. 1, WITH-loops in SaC_λ take the general form

with `(` *lower* `<=` *idx_vec* `<` *upper*`)` `:` *expr*
genarray`(` *shape*, *default*`)`

where *idx_vec* is an identifier, *lower*, *upper*, and *shape* denote expressions that should evaluate to vectors of identical length and *expr* and *default* denote arbitrary expressions that need to evaluate to arrays of identical shape. Such a WITH-loop defines an array of shape *shape*, whose elements are either computed from the expression *expr* or from the default expression *default*. Which of these two values is chosen for an individual element depends on the element's location, i.e., it depends on its index position. If the index is within the range specified by the lower bound *lower* and the upper bound *upper*, *expr* is chosen, otherwise *default* is taken. As a simple example, consider the WITH-loop

```
with ([1] <= iv < [4]) : 2
genarray( [5], 0)
```

It computes the vector `[0, 2, 2, 2, 0]`. Note here, that the use of vectors for the shape of the result and the bounds of the index space (also referred to as the "generator"') allows WITH-loops to denote arrays of arbitrary rank. Furthermore, the "generator expression" *expr* may refer to the index position through the "generator variable" *idx_vec*[1]. For example, the WITH-loop

[1] Most of our examples use `iv` as variable name for the generator variable.

```
with ([1,1] <= iv < [3,4]) : sel([0], iv) + sel([1], iv)
genarray( [3,5], 0)
```

yields the matrix $\begin{pmatrix} 0\,0\,0\,0\,0 \\ 0\,2\,3\,4\,0 \\ 0\,3\,4\,5\,0 \end{pmatrix}$.

We can formalise the semantics of SaC$_\lambda$ by a standard big-step operational semantics for λ-calculus-based applicative languages as defined in several textbooks, e.g., [Pie02]. The core relations, i.e., those for conditionals, abstractions, and function applications can be used in their standard form. Hence, only those relations pertaining to the array specific features of SaC$_\lambda$ are shown in Fig. 2.

$$\text{CONST}: \frac{}{n \Downarrow < [], [n] >}$$

$$\text{VECT}: \frac{\forall i \in \{1,\ldots,n\} : e_i \Downarrow < [\, s_1, \ldots, s_m], [\, d_1^i, \ldots, d_p^i] >}{[\, e_1, \ldots, e_n] \Downarrow < [\, \mathrm{n},\ s_1, \ldots, s_m], [\ d_1^1, \ldots, d_p^1, \ldots,\ d_1^n, \ldots, d_p^n] >}$$

$$\text{DIM}: \frac{e \Downarrow < [\, s_1, \ldots, s_n], [\, d_1, \ldots, d_m] >}{\texttt{dim(}\ e\texttt{)} \Downarrow < [], [n] >}$$

$$\text{SHAPE}: \frac{e \Downarrow < [\, s_1, \ldots, s_n], [\, d_1, \ldots, d_m] >}{\texttt{shape(}\ e\texttt{)} \Downarrow < [\, n], [\, s_1, \ldots, s_n] >}$$

$$\text{SEL}: \frac{\begin{array}{c} iv \Downarrow < [\, n], [\, i_1, \ldots, i_n] > \\ e \Downarrow < [\, s_1, \ldots, s_n], [\, d_1, \ldots, d_m] > \end{array}}{\begin{array}{c} \texttt{sel(}\ iv\texttt{,}\ e\texttt{)} \Downarrow < [], [\, d_l] > \\ \text{where } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) \\ \iff \forall k \in \{1,\ldots,n\} : 0 \le i_k < s_k \end{array}}$$

$$*: \frac{\begin{array}{c} e_1 \Downarrow < [\, s_1, \ldots, s_n], [\, d_1^1, \ldots, d_m^1] > \\ e_2 \Downarrow < [\, s_1, \ldots, s_n], [\, d_1^2, \ldots, d_m^2] > \end{array}}{\texttt{*(}\ e_1\texttt{,}\ e_2\texttt{)} \Downarrow < [\, s_1, \ldots, s_n], [\, d_1^1 * d_1^2, \ldots, d_m^1 * d_m^2] >}$$

$$\text{WITH}: \frac{\begin{array}{c} e_l \Downarrow < [\, n], [\, l_1, \ldots, l_n] > \\ e_u \Downarrow < [\, n], [\, u_1, \ldots, u_n] > \\ e_{shp} \Downarrow < [\, n], [\, shp_1, \ldots, shp_n] > \\ e_{def} \Downarrow < [\, s_1, \ldots, s_m], [\, d_1, \ldots, d_p] > \\ \forall i_1 \in \{l_1, ..., u_1 - 1\} \ldots \forall i_n \in \{l_n, ..., u_n - 1\} : (\lambda\, Id\texttt{.}\, e_b\ \texttt{[}\ i_1\texttt{,}\ \ldots\texttt{,}\ i_n\texttt{]}) \\ \Downarrow < [\, s_1, \ldots, s_m], [\, d_1^{[i_1,\ldots,i_n]}, \ldots, d_p^{[i_1,\ldots,i_n]}] > \end{array}}{\begin{array}{c} \texttt{with(}\ e_l\ \texttt{<=}\ Id\ \texttt{<}\ e_u\texttt{)}\ \texttt{:}\ e_b\ \texttt{genarray(}\ e_{shp}\texttt{,}\ e_{def}\texttt{)} \\ \Downarrow < [\ shp_1, \ldots, shp_n,\ s_1, \ldots, s_m], \\ [\ d_1^{[0,\ldots,0]}, \ldots, d_p^{[0,\ldots,0]}, \ldots,\ d_1^{[shp_1-1,\ldots,shp_n-1]}, \ldots, d_p^{[shp_1-1,\ldots,shp_n-1]}] > \\ \text{where } d_i^{[x_1,\ldots,x_n]} = d_i \text{ iff } \exists j \in \{1,...,n\} : x_j \in \{0,...,l_j - 1\} \cup \{u_j,...,shp_j - 1\} \end{array}}$$

Fig. 2. An operational semantics for SaC$_\lambda$

As a unified representation for n-dimensional arrays we use pairs of vectors $< [\ shp_1, \ldots, shp_n], [\ data_1, \ldots, data_m] >$ where the vector $[\ shp_1, \ldots, shp_n]$ denotes the shape of the array, i.e., its extent with respect to the n individual axes, and the vector $[\ data_1, \ldots, data_m]$ contains all elements of the array in a linearised form. Since the number of elements within an array equals the product of the number of elements per individual axes, we have $m = \prod_{i=1}^{n} shp_i$.

The first two evaluation rules of Fig. 2 show how scalars as well as vectors are transformed into the internal representation. Note with the rule VECT, that all elements need to be of the same shape which ensures shape consistency in the overall result.

The next three rules formalise the semantics of the main primitive operations on arrays: `dim`, `shape`, and `sel`. Element-wise extensions of standard operations such as the arithmetic and relational operations are demonstrated by the example of the rule for multiplication (`*`).

The last rule gives the formal semantics of the WITH-loop in SaC$_\lambda$. The first three conditions require the lower bound, the upper bound and the shape expression to evaluate to vectors of identical length. The next two conditions relate to the default expression e_{def} and the generator expression e_b, respectively. They ensure, that the default expression evaluates to an array of the same shape as the generator expression does. Since the generator expression may refer to the index variable, this is formalised by transforming the generator expression into an anonymous function and by evaluating a pseudo-application of this function to all indices specified in the generator. The lower part of the WITH-loop-rule shows how the values from the individual generator expression evaluations and the value of the default expression are combined into the overall result. The shape of the result stems from concatenating the shape expression with the shape of the default element. Its data vector consists of a concatenation of the data vectors from the individual generator expression evaluations. Since the generator does not necessarily cover the entire index space, the default expression values need to be inserted whenever at least one element of the index vector $[i_1, \ldots, i_n]$ is outside the generator range, i.e., $\exists j \in \{1, ..., n\} : x_j \in \{0, ..., l_j - 1\} \cup \{u_j, ..., shp_j - 1\}$. Formally this is achieved by the "where clause" of the rule WITH.

3 A Motivating Example

The core language introduced in the previous section suffices to define generic array operations similar to those available in array languages such as APL, NIAL, or J. As an example, consider the operations `take` and `create` as defined in Fig. 3. The function `take` expects two arguments `v` and `a`. It returns an array of shape `v` whose elements are copied from those in the corresponding positions of the argument array `a`. Note here, that the specification of `0*v` as lower bound yields a vector of zeros of the same length as the vector `v` and, thus, ensures shape-invariance, i.e., it makes `take` applicable to arrays of arbitrary dimensionality.

```
take = λv,a.with ( 0*v <= iv < v): sel( iv, a)
             genarray( v, 0)
create = λs,x.with ( 0*s <= iv < s): x
               genarray( s, x)
```

Fig. 3. A definition of `take` and `create` in SaC$_\lambda$

The function `create` takes two arguments as well: a shape vector `s` and a value `x`. From these it computes an array of shape `s` with all elements identical to `x`. Again, shape-invariance is achieved by computing the bounds from the vector `s` that determines the shape of the result.

Both these functions are non-uniform, i.e., result shapes cannot be computed from the argument shapes only. Instead, argument values are required to determine the result shapes. Several application studies show that functions of this sort usually prove very useful when adopting a compositional programming style [Sch03, GS99]. A typical application of these operations is shown in Fig. 4. The

```
matmul = λdl,dm,v.let
                     maind = dm * v
                  in let
                       lowerd = dl * take( shape( dl), v)
                     in let
                          zeros = create( shape( dm) - shape( dl), 0)
                        in maind + concat( zeros, lowerd)
```

Fig. 4. A definition of a sparse matrix vector multiply in SaC$_\lambda$

function `matmul` implements a special case for a matrix vector product where the matrix contains non-zero values on two diagonals only: the main diagonal (argument `dm`) and another diagonal `dl` located below the main one. A third argument `v` represents the vector the matrix is to be multiplied with and, thus, is expected to have as many elements as the main diagonal `dm` does. The difference in length between the two diagonals determines the exact location of the lower diagonal. Essentially, the matrix vector product consists of the sum of products `dm * v` and `dl * v`. However, the vector `v` needs to be shortened prior to the multiplication with `dl` to match its size, and the resulting vector (`lowerd` in Fig. 4) needs to be prepended by sufficient zeros in order to match the length of the main diagonal `dm`. The latter is achieved by concatenating a vector of zeros (`zeros` in Fig. 4) of appropriate length.

The most remarkable aspect of this function is that although it makes use of the two non-uniform operations `take` and `create`, `matmul` itself is uniform. This stems from the fact that the shape determining arguments of `take` and `create` are computed from the shape of the arguments `dm` and `dl`, a programming pattern that can be observed rather frequently.

A brute force approach to static inference based on specialisation to argument shapes only would only yield the dimensionalities for the results of the applications of `take` and `create`, not their shapes. This, in turn, would lead to the loss of static result shape knowledge for `matmul` itself. That knowledge can only be gained, if `take` and `create` both are specialised wrt. values in their first argument position, and if the subtraction in the first argument position of create is computed statically.

The overall goal of the analysis presented in this paper is to statically infer to what extent functions need to be specialised in order to achieve a certain level of information for their results. In the given example, the analysis should yield that `take` and `create` need to be specialised to values if the result shape is required, and that for `matmul` it suffices to specialise wrt. argument shapes. However, the analysis should also yield to which extent all subexpressions need to be calculated statically in order to achieve that goal.

4 Basic Approach

Traditionally, binding time analysis is based on a two element domain: all expressions are either attributed as *static* or as *dynamic*. In our approach, we distinguish four different levels of static array information[2]:

AUD (**A**rray of **U**nknown **D**imensionality):
no shape information is available at all;
AKD (**A**rray of **K**nown **D**imensionality):
dimensionality is known but not the exact shape;
AKS (**A**rray of **K**nown **S**hape):
the exact shape is available at compile-time;
AKV (**A**rray of **K**nown **V**alue):
not only the exact shape but also the value is statically known.

These four levels build the grounds for our analysis. We try to infer to which extent static knowledge of the arguments of a function is needed in order to achieve a certain level of static information about the result. Although we are primarily interested in the level of information that is required for statically computing the shape of the result only (AKS result), we need to infer the required levels for all possible result levels. This extended effort is required as we may find function applications in positions where other levels of shape information than just AKS are required. Consider, for example, the expression `shape( dm) - shape( dl)` of the `matmul` example. Here, it is essential for the inference to find out which level of information is required for `dm` and `dl` in order to compute the value of the expression statically.

As a consequence, we do not attribute each expression with one of these levels only, but we need to infer mappings from the set of levels {AUD, AKD, AKS,

[2] Readers familiar with SAC may notice that these levels directly correspond to the hierarchy of array types in SAC which is essential when it comes to implementing the specialisation phase.

AKV} into itself. Once we have inferred such mappings for all arguments of a function, we can use this information to find out which level of specialisation is required in order to achieve a certain level of result information.

Let us consider the built-in operation `shape` as an example. For its relation between result level and argument level, we find the following mapping in our four-element-domain:

$$\begin{array}{l}\{\ \mathrm{AUD}\rightarrow\mathrm{AUD},\\ \ \ \mathrm{AKD}\rightarrow\mathrm{AUD},\\ \ \ \mathrm{AKS}\rightarrow\mathrm{AKD},\\ \ \ \mathrm{AKV}\rightarrow\mathrm{AKS}\}\end{array}$$

As the result of the primitive function `shape` always is a vector, no array information at all is needed if we are interested in the dimensionality of the result. The shape of the result requires the dimensionality of the argument only, and the value of the result can be deduced from the shape of the argument.

In order to formalise this approach, we can identify the different levels of array information as coarsening steps in the value domain of SaC_λ. While AKV is identical to our original domain of values of the form

$$< [\ shp_1, \ldots, shp_n], [\ data_1, \ldots, data_m] >,$$

AKS can be described by values of the form

$$< [\ shp_1, \ldots, shp_n], - >.$$

Taking the use of the '–' symbol for irrelevant values further, we can use

$$< [\ \overbrace{-, \ldots, -}^{n}], - >$$

for AKD arrays and

$$< -, - >$$

for AUD arrays.

With these new domains, we can now deduce new semantic rules from those of Fig. 2. We successively weaken the preconditions to less precise domains and determine the effect of this information loss on the postconditions. Applying this approach to the SHAPE-rule, we obtain three new rules:

$$\mathrm{AKS}_{\mathrm{SHAPE}}: \frac{e \Downarrow < [\ s_1, \ldots, s_n], - >}{\texttt{shape(}\ e\texttt{)} \Downarrow < [\ n], [\ s_1, \ldots, s_n] >}$$

$$\mathrm{AKD}_{\mathrm{SHAPE}}: \frac{e \Downarrow < [\ \overbrace{-, \ldots, -}^{n}], - >}{\texttt{shape(}\ e\texttt{)} \Downarrow < [\ n], - >}$$

$$\mathrm{AUD}_{\mathrm{SHAPE}}: \frac{e \Downarrow < -, - >}{\texttt{shape(}\ e\texttt{)} \Downarrow < [\ -], - >}$$

From these rules we observe that

- AKS arguments are mapped into AKV ones
- AKD arguments are mapped into AKS ones
- AUD arguments are mapped into AKD ones

As we are interested in predicting the required argument shape-levels for a desired return shape-level, we are actually looking for the inverse of the mapping deduced from the semantic rules. The inverse is well-defined as all functions are monotonic with respect to the array information hierarchy, i.e., providing more shape information can never lead to fewer shape information of the result. Furthermore, the finite domain/codomain guarantees an effective computability of the inverse.

In case of the `shape` operation, we obtain exactly the same mapping as the one we have derived earlier in an informal fashion.

Uniformity of a function can now easily be recognised from its associated mapping: whenever AKS is mapped into a shape-level less or equal to AKS, we know that the shape of the function's result does at most require the shape of the argument, not its value. The unpleasant non-uniform cases are those where AKS is mapped into AKV.

5 Towards an Inference Algorithm

Rather than just giving the coarsened semantic rules, in the sequel, we develop an algorithm for effectively inferring the shape-level mappings described in the previous section for arbitrary SaC$_\lambda$ programs.

In order to achieve a more concise notation, we encode our four-element-domain by the numbers 0,1,2 and 3. This allows us to represent the mappings on that domain as four-element-vectors of these numbers. Applications of these mappings then boil down to selections into the vector. Using 0 for AUD, 1 for AKD, 2 for AKS, and 3 for AKV, we can encode the mapping for `shape` as $[0, 0, 1, 2]$. Similarly, we obtain the vector $[0, 0, 0, 1]$ for the primitive operation `dim`. It shows that only if we are interested in the result value itself we need to know something about the argument and all we need to know is its dimensionality.

We refer to these vectors as *propagation vectors* as they, for a given function application, propagate a given return value demand into a demand for the arguments. If we are, for example, interested in the value of an expression `shape( dm)`, i.e., we have a demand of 3 (AKV), this demand propagates into a demand on `dm` by selecting the third element of the propagation vector of `shape` yielding [0,0,1,2][3] = 2 (AKS) as demand for `dm`.

Functions with more than just one argument require as many propagation vectors as we have arguments. For example, the built-in selection operation `sel` has two propagation vectors: $\begin{bmatrix}[0, 2, 2, 3] \\ [0, 1, 2, 3]\end{bmatrix}$. If we are interested in the dimensionality of the result, we need to consult the second element in each propagation vector. It shows that the shape of the selection vector (first argument) is needed as well as the dimensionality of the array to be selected from (second argument).

Computing propagation vectors of entire functions essentially boils down to propagating all four possible demands through the body expression and collecting the resulting demands for the individual arguments as vectors. As an example, let us consider the expression λ`a.sel([0], shape( shape( a)))`. It

computes the shape of the shape of an array `a` and selects the only component of the resulting vector which is identical to computing the array's dimensionality. Hence, we expect a propagation vector identical to that of the primitive operation `dim` to be computed for this function.

First, let us compute the demand for `a` assuming we need to statically compute the value of the overall expression, i.e., we have an initial demand of 3 (AKV). That demand propagates into the second argument of the selection by applying the second propagation vector of `sel` to it, i.e., we obtain [0,1,2,3][3] = 3 (AKV) as demand for the subexpression `shape( shape( a))`. Propagating that demand through the outer application of `shape` yields [0,0,1,2][3] = 2 (AKS) which subsequently is propagated through the inner application of `shape` resulting in [0,0,1,2][2] = 1 (AKD) as demand for `a`.

Similarly, the other three possible overall demands can be propagated through the function body. All these result in a demand of 0 (AUD) for `a`. Combining these results into a vector yields [0,0,0,1] as propagation vector for the given function which corresponds to the propagation vector of the built-in operation `dim`.

As all four demands can be computed independently, the propagation in fact can be implemented as a data parallel operation that propagates entire demand vectors through the function bodies, starting out from the canonical demand vector [0,1,2,3].

6 Inferring Propagation Vectors

So far, all our example functions were combinators, i.e., they did not contain any relatively free variables. Although that holds for all built-in operators and for all user-defined functions in SaC$_\lambda$, it does not hold for arbitrary expressions. These can be nested let-expressions or WITH-loops both of which introduce locally scoped variables. To address this situation, any inference scheme for propagation vectors needs to deal with environments that hold demands for relatively free variables.

We introduce a scheme $\mathcal{SD}(expr,\ dem,\ \mathcal{F})$ which computes an environment that contains demand vectors for all relatively free variables of an expression *expr*. It expects two additional parameters: an overall demand *dem*, and a function environment $\mathcal{F}$ that contains the propagation vectors of all functions. Fig. 5 shows a formal definition of that scheme. Constants meet any demand and do not raise any new demands, hence, an empty set is returned for constants. If a given demand *dem* is imposed on a variable Id then the singleton set is returned containing the pair of the identifier and the given demand.

For function applications, the demand is translated into argument demands by the appropriate propagation vectors first. These are either extracted from the function environment $\mathcal{F}$, or — in case of built-in operators — they are determined by an auxiliary scheme $\mathcal{PV}$. After the initial demand *dem* has been translated into demands dem_i for the individual arguments, the scheme is recursively applied to the argument expressions. The resulting sets of demands

$$\mathcal{SD}(Const,\ dem,\ \mathcal{F}) = \{\}$$

$$\mathcal{SD}(Id,\ dem,\ \mathcal{F}) = \{Id : dem\}$$

$$\mathcal{SD}(FunId(e_1, ..., e_n),\ dem,\ \mathcal{F}) = \bigoplus_{i=1}^{n} \mathcal{SD}(e_i,\ dem_i,\ \mathcal{F})$$
where $dem_i = (\mathcal{F}(FunId)_i)[dem]$

$$\mathcal{SD}(Prf(e_1, ..., e_n),\ dem,\ \mathcal{F}) = \bigoplus_{i=1}^{n} \mathcal{SD}(e_i,\ dem_i,\ \mathcal{F})$$
where $dem_i = (\mathcal{PV}(Prf)_i)[dem]$

$$\mathcal{SD}(\texttt{let}\ Id = e_1 \texttt{in}\ e_2,\ dem,\ \mathcal{F}) = \begin{pmatrix} (\ \mathcal{SD}(e_2,\ dem,\ \mathcal{F}) \setminus \{Id\}\ ) \\ \oplus\ \mathcal{SD}(e_1,\ dem',\ \mathcal{F}) \end{pmatrix}$$
where $dem' = \mathcal{PV}(\lambda\, Id\,\texttt{.}\, e_2)[dem]$

$$\mathcal{SD}\left(\begin{matrix}\texttt{with}(e_{lb}\texttt{<=}Id\texttt{<}e_{ub}) : e \\ \texttt{genarray}(e_{shp}, e_{def})\end{matrix},\ dem,\ \mathcal{F}\right) = \begin{pmatrix} \mathcal{SD}(e_{shp},\ dem_s,\ \mathcal{F}) \\ \oplus\ (\ \mathcal{SD}(e,\ dem,\ \mathcal{F}) \setminus \{Id\}\ ) \\ \oplus\ \mathcal{SD}(e_{def},\ dem,\ \mathcal{F}) \\ \oplus\ \mathcal{SD}(e_{lb},\ dem_{Id},\ \mathcal{F}) \\ \oplus\ \mathcal{SD}(e_{ub},\ dem_{Id},\ \mathcal{F}) \end{pmatrix}$$
where $dem_s = [0,2,3,3][dem]$
$dem_{Id} = \mathcal{PV}(\lambda\, Id\,\texttt{.}\,\mathrm{e})[dem]$

Fig. 5. Scheme for inferring specialisation demands

for relatively free variables are combined by an operation denoted as $\oplus$. It constitutes a union of sets for those variables that occur in one set only and an element-wise maximum on the demand vectors for all variables that occur in both sets.

Let-expressions essentially are a combination of the demands in the body and the demands in the defining expression. However, the external demand *dem* needs to be translated into the demand for the defining expression dem' by computing the propagation vector for the underlying λ-abstraction. Furthermore, we need to exclude the demand for the defined variable from the demands inferred from the body of the let-expression as relatively free occurrences in the body relate to this very definition.

The dominating rule for inferring specialisation demands of array operations is the rule for WITH-loops as these are the predominant language constructs for defining array operations in SaC. While the overall demand *dem* can be propagated without modification into the generator expression e and the default expression e_{def}, the most important effect is the increase in demand for the shape expression e_{shp}. Here, we have a propagation vector [0,2,3,3] which indicates that we lose one level of shape information. As a consequence, we need

to statically infer the exact value of this expression if we want to find out the shape of the result. The overall demand of the WITH-loop, again, is the combination of the demands of the individual components using the translated demands dem_s, dem_e, and dem_{Id} for the shape expression, defining expressions, and the boundary expressions, respectively.

All that remains to be defined is the auxiliary scheme for obtaining the propagation vectors $\mathcal{PV}$ as shown in Fig. 6. It takes a function and returns a vector

$$\mathcal{PV}(shape) = [[0,0,1,2]]$$

$$\mathcal{PV}(dim) = [[0,0,0,1]]$$

$$\mathcal{PV}(sel) = \begin{bmatrix} [0,2,2,3] \\ [0,1,2,3] \end{bmatrix}$$

$$\mathcal{PV}(*) = \begin{bmatrix} [0,1,2,3] \\ [0,1,2,3] \end{bmatrix}$$

$$\mathcal{PV}(\lambda\, Id_1, ..., Id_n \,.\, \mathrm{e}) = \begin{bmatrix} \mathcal{SD}(e,\ [0,1,2,3],\ \mathcal{F})(Id_1) \\ \vdots \\ \mathcal{SD}(e,\ [0,1,2,3],\ \mathcal{F})(Id_n) \end{bmatrix}$$

Fig. 6. Computing propagation vectors

of propagation vectors. For built-in operations such as `shape`, `dim`, etc. these are constants defined as explained earlier. For user defined functions or abstract functions as introduced by the scheme $\mathcal{SD}$, the scheme $\mathcal{SD}$ itself can be utilised. It is applied to the body of the function, assuming demand for all four different levels ([0,1,2,3]). As this yields the demands for all relatively free variables it suffices to select those entries that relate to the binding λ. Variables that do not occur in these sets are not used within the body and, thus, obtain the propagation vector [0,0,0,0]. This is realised by the selection operation denoted as $\mathcal{SD}(...)(Id_i)$.

With these definitions, we can define the overall propagation vector environment for user-defined functions $\mathcal{F}$. Assuming a program of the form

$$\begin{aligned} f_1 &= e_1 \\ &\vdots \\ f_n &= e_n \\ \texttt{main} &= e \end{aligned}$$

we obtain:

$$\mathcal{F} = \bigoplus_{i=1}^{n} \{ f_i : \mathcal{PV}(e_i) \} \quad .$$

The interesting aspect of this definition, from an implementational point of view, is its recursive nature which arises from the reference to $\mathcal{F}$ in the definition of $\mathcal{PV}(e_i)$. However, due to the monotonicity of the maximum of the $\oplus$ operation and the finiteness of the domain, the computation of $\mathcal{F}$ can be implemented as a fixed-point iteration starting with propagation vectors [0,0,0,0].

7 Applying the Inference Algorithm

This section illustrates the formalism of the previous section by providing a formal derivation of the propagation for the functions `take` and `matmul` from Section 2. For `take`, we obtain:

$$\mathcal{PV}(\lambda\,\mathtt{v,a}.\,body_{take}) \;=\; \begin{bmatrix} \mathcal{SD}(body_{take},\ [0,1,2,3],\ \mathcal{F})(\mathtt{v}) \\ \mathcal{SD}(body_{take},\ [0,1,2,3],\ \mathcal{F})(\mathtt{a}) \end{bmatrix}$$

Propagating the canonical demand [0,1,2,3] into the body of `take`, we obtain demands for the subexpressions of the WITH-loop:

$$\begin{aligned} &\mathcal{SD}(body_{take},\ [0,1,2,3],\ \mathcal{F}) \\ &= \mathcal{SD}\left(\begin{matrix} \mathtt{with(0 * v <= iv < v) : sel(iv, a)} \\ \mathtt{genarray(v, 0)} \end{matrix},\ [0,1,2,3],\ \mathcal{F} \right) \\ &= \begin{pmatrix} \quad \mathcal{SD}(\mathtt{v},\ [0,1,2,3],\ \mathcal{F}) \\ \oplus\ (\ \mathcal{SD}(\mathtt{sel(iv, a)},\ [0,1,2,3],\ \mathcal{F}) \setminus \{\mathtt{iv}\}\ ) \\ \oplus\ \mathcal{SD}(\mathtt{0},\ [0,1,2,3],\ \mathcal{F}) \\ \oplus\ \mathcal{SD}(\mathtt{0 * v},\ \mathcal{PV}(\lambda\,\mathtt{iv.sel(\ iv,\ a)})[0,1,2,3],\ \mathcal{F}) \\ \oplus\ \mathcal{SD}(\mathtt{v},\ \mathcal{PV}(\lambda\,\mathtt{iv.sel(\ iv,\ a)})[0,1,2,3],\ \mathcal{F}) \end{pmatrix} \end{aligned}$$

The demand for the lower and upper bound expressions of the generator of the WITH-loop is computed as demand for `iv` when propagating the actual demand through the body expression `sel( iv, a)`. This is done by first computing the propagation vector for the pseudo-function $\lambda\,$`iv.sel( iv, a)`:

$$\begin{aligned} &\mathcal{PV}(\lambda\,\mathtt{iv.sel(\ iv,\ a)}) \\ &= [\mathcal{SD}(\mathtt{sel(iv, a)},\ [0,1,2,3],\ \mathcal{F})(\mathtt{iv})] \\ &= [\{\mathtt{iv} : [0,2,2,3]\}(\mathtt{iv})] \\ &= [[0,2,2,3]] \end{aligned}$$

With this propagation the demand for the bounds can be computed by mapping the actual demand [0,1,2,3] on a selection into [0,2,2,3] which yields [0,2,2,3]. With this demand we obtain

$$\begin{aligned} &\mathcal{SD}(\mathtt{v},\ \mathcal{PV}(\lambda\,\mathtt{iv.sel(\ iv,\ a)})[0,1,2,3],\ \mathcal{F}) \\ &= \mathcal{SD}(\mathtt{v},\ [[0,2,2,3]],\ \mathcal{F}) \\ &= \{\mathtt{v} : [0,2,2,3]\} \end{aligned}$$

and
$\mathcal{SD}(\texttt{0} * \texttt{v},\ \mathcal{PV}(\lambda\,\texttt{iv.sel(iv, a)})[0,1,2,3],\ \mathcal{F}) = \{\texttt{v} : [0,2,2,3]\}$

For the result shape we have
$\mathcal{SD}(\texttt{v},\ [0,2,3,3],\ \mathcal{F}) = \{\texttt{v} : [0,2,3,3]\}$.

From the body expression a demand on `a` arises as
$(\ \mathcal{SD}(\texttt{sel}(\texttt{iv},\texttt{a}),\ [0,1,2,3],\ \mathcal{F}) \setminus \{\texttt{iv}\}\ ) = \{\texttt{a} : [0,1,2,3]\}$.

As the default expression is constant we have
$\mathcal{SD}(\texttt{0},\ [0,1,2,3],\ \mathcal{F}) = \{\}$.

Taking these together, we eventually obtain
$\mathcal{SD}(body_{take},\ [0,1,2,3],\ \mathcal{F}) = \{\texttt{v} : [0,2,3,3], \texttt{a} : [0,1,2,3]\}$

which gives
$$\mathcal{PV}(\lambda\,\texttt{v,a}\,.\,body_{take}) = \begin{bmatrix}[0,2,3,3]\\ [0,1,2,3]\end{bmatrix}.$$

From this result, we can easily identify the non-uniformity in the first argument position of `take`. If the shape of the result is required, the demand of the individual arguments can be derived from the third position in the propagation vectors. They show that we do need to specialise the first argument wrt. to the argument value while it suffices to specialise the second argument wrt. its shape. Similarly, we obtain for `create`:
$$\mathcal{PV}(\lambda\,\texttt{s,x}\,.\,body_{take}) = \begin{bmatrix}[0,2,3,3]\\ [0,1,2,3]\end{bmatrix}.$$

Having these in place, we can now infer the propagation for `matmul`:
$$\mathcal{PV}(\lambda\,\texttt{dl,dm,v}\,.\,body_{mm})\ \texttt{=}\ \begin{bmatrix}\mathcal{SD}(body_{mm},\ [0,1,2,3],\ \mathcal{F})\texttt{(dl)}\\ \mathcal{SD}(body_{mm},\ [0,1,2,3],\ \mathcal{F})\texttt{(dm)}\\ \mathcal{SD}(body_{mm},\ [0,1,2,3],\ \mathcal{F})\texttt{(v)}\end{bmatrix}$$

Propagating the canonical demand [0,1,2,3] into the body of `matmul` we obtain:
$\mathcal{SD}(body_{mm},\ [0,1,2,3],\ \mathcal{F})$
$$= \mathcal{SD}\left(\begin{array}{l}\texttt{let}\\ \quad \texttt{maind} = \texttt{dm} * \texttt{v},\ [0,1,2,3],\ \mathcal{F}\\ \texttt{in}\ letbody\end{array}\right)$$
$$= \left(\begin{array}{l}\quad \mathcal{SD}(letbody,\ [0,2,3,3],\ \mathcal{F})\\ \oplus\ \mathcal{SD}(\texttt{dm} * \texttt{v},\ \mathcal{PV}(\lambda\,\texttt{maind}\,.\,letbody)[0,1,2,3],\ \mathcal{F})\end{array}\right)$$

Since $\mathcal{PV}(\lambda\,\texttt{maind}\,.\,letbody) = [\mathcal{SD}(letbody,\ [0,1,2,3],\ \mathcal{F})(\texttt{maind})]$ we can see how the inference is driven bottom-up. Computing $\mathcal{SD}(letbody,\ [0,1,2,3],\ \mathcal{F})$ recursively leads us into the innermost goal expression, i.e., `maind + concat( zeros, lowerd)`. As both, addition and concatenation are uniform, we have
$\mathcal{SD}(\texttt{maind} + \texttt{concat}(\texttt{zeros},\texttt{lowerd}),\ [0,1,2,3],\ \mathcal{F})$
$= \{\texttt{maind} : [0,1,2,3], \texttt{zeros} : [0,1,2,3], \texttt{lowerd} : [0,1,2,3]\}$

From this, we obtain that
$\mathcal{PV}(\lambda\,\texttt{zeros.maind + concat(zeros, lowerd)})[0,1,2,3] = [0,1,2,3]$

and thus

$$\mathcal{SD}\left(\begin{array}{l}\texttt{let}\\ \quad\texttt{zeros = create(shape(dm) - shape(dl), 0)},\ [0,1,2,3],\ \mathcal{F}\\ \texttt{in maind + concat(zeros, lowerd)}\end{array}\right)$$
$$= \left(\begin{array}{l}\{\texttt{maind} : [0,1,2,3], \texttt{zeros} : [0,1,2,3], \texttt{lowerd} : [0,1,2,3]\}\\ \oplus\mathcal{SD}(\texttt{create}(\texttt{shape}(\texttt{dm}) - \texttt{shape}(\texttt{dl}), \texttt{0}),\ [0,1,2,3],\ \mathcal{F})\end{array}\right).$$

Here, we have reached the most interesting aspect of the inference for `matmul`. Although `create` is non-uniform, we expect this expression not to raise a demand higher than [0,1,2,3] for the variables `dm` and `dl`. Following the inference algorithm rules, we obtain:

$\mathcal{SD}(\texttt{create}(\texttt{shape}(\texttt{dm}) - \texttt{shape}(\texttt{dl}), \texttt{0}),\ [0,1,2,3],\ \mathcal{F})$
$= \mathcal{SD}(\texttt{shape}(\texttt{dm}) - \texttt{shape}(\texttt{dl}),\ [0,2,3,3],\ \mathcal{F})$

as the constant `0` does not raise any demand. As subtraction is uniform the demand that was raised to [0,2,3,3] by `create` is propagated into the individual subexpression, i.e., we have

$\mathcal{SD}(\texttt{shape}(\texttt{dm}) - \texttt{shape}(\texttt{dl}),\ [0,2,3,3],\ \mathcal{F})$
$= \mathcal{SD}(\texttt{shape}(\texttt{dm}),\ [0,2,3,3],\ \mathcal{F}) \oplus \mathcal{SD}(\texttt{shape}(\texttt{dl}),\ [0,2,3,3],\ \mathcal{F})$

According to the rule for primitive functions, we obtain as demand for `dm` as well as `dl`: $[0,0,1,2][[0,2,3,3]] = [0,1,2,2]$. From this result, we can see that we obtain a demand of [0,1,2,2] which is even lower than the expected demand [0,1,2,3]. Having a closer look at the expression, we can observe that the value of the entire expression in fact does not depend on the values of `dm` and `dl` but their shapes only.
Using this result, we obtain

$$\mathcal{SD}\left(\begin{array}{l}\texttt{let}\\ \quad\texttt{zeros = create(shape(dm) - shape(dl), 0)},\ [0,1,2,3],\ \mathcal{F}\\ \texttt{in maind + concat(zeros, lowerd)}\end{array}\right)$$
$$= \left(\begin{array}{l}\{\texttt{maind} : [0,1,2,3], \texttt{zeros} : [0,1,2,3], \texttt{lowerd} : [0,1,2,3]\}\\ \oplus\{\texttt{dm} : [0,1,2,2], \texttt{dl} : [0,1,2,2]\}\end{array}\right).$$

Propagating that information further up, we obtain

$\mathcal{SD}(\mathit{letbody},\ [0,1,2,3],\ \mathcal{F})$
$$= \mathcal{SD}\left(\begin{array}{l}\texttt{let}\\ \quad\texttt{lowerd = dl * take(shape(dl), v)},\ [0,1,2,3],\ \mathcal{F}\\ \texttt{in}\ \mathit{letbody2}\end{array}\right)$$
$$= \left(\begin{array}{l}\{\texttt{maind} : [0,1,2,3], \texttt{dm} : [0,1,2,2], \texttt{dl} : [0,1,2,2]\}\\ \oplus\ \mathcal{SD}(\texttt{dl} * \texttt{take}(\texttt{shape}(\texttt{dl}), \texttt{v}),\ [0,1,2,3],\ \mathcal{F})\end{array}\right).$$

As we have

$\mathcal{SD}(\texttt{dl} * \texttt{take}(\texttt{shape}(\texttt{dl}), \texttt{v}),\ [0,1,2,3],\ \mathcal{F}) = \{\texttt{dl} : [0,1,2,3], \texttt{v} : [0,1,2,3]\}$

we further obtain

$$\begin{aligned}&\mathcal{SD}(letbody,\ [0,1,2,3],\ \mathcal{F})\\&= \{\texttt{maind} : [0,1,2,3], \texttt{dm} : [0,1,2,2], \texttt{dl} : [0,1,2,3], \texttt{v} : [0,1,2,3]\}\end{aligned}$$

Note here how the multiplication with `dl` increases the overall demand for that variable in the AKV case from AKS to AKV.
Eventually, we obtain for the entire body of `matmul`:

$$\begin{aligned}&\mathcal{SD}(body_{mm},\ [0,1,2,3],\ \mathcal{F})\\&= \{\texttt{dm} : [0,1,2,2], \texttt{dl} : [0,1,2,3], \texttt{v} : [0,1,2,3]\} \oplus \mathcal{SD}(\texttt{dm} * \texttt{v},\ [0,1,2,3],\ \mathcal{F})\\&= \{\texttt{dm} : [0,1,2,2], \texttt{dl} : [0,1,2,3], \texttt{v} : [0,1,2,3]\} \oplus \{\texttt{dm} : [0,1,2,3], \texttt{v} : [0,1,2,3]\}\\&= \{\texttt{dm} : [0,1,2,3], \texttt{dl} : [0,1,2,3], \texttt{v} : [0,1,2,3]\}\end{aligned}$$

Similar as with `dl`, the use of `dm` as factor increases the demand for `dm`. This supports our intuitive result that `matmul` is a uniform function with

$$\mathcal{PV}(\lambda\,\texttt{dl,dm,v}.\,body_{mm}) = \begin{bmatrix}[0,1,2,3]\\[0,1,2,3]\\[0,1,2,3]\end{bmatrix}.$$

8 Related Work

Generic programming on arrays can also be found in the programming language Fish [JMB98, JS98]. It is based on the idea to divide up all functions into two parts: one part that describes the actual computation of values and another part that describes the computation of result shapes from argument shapes. While the former is implemented at runtime, the latter is done statically by the compiler. This separation eases the specialisation as the static parts are identified by the programmer. In fact, it can be considered an offline approach to partially evaluating Fish programs. However, specialisation wrt. argument values in Fish cannot happen since all shape computations need to be defined in terms of argument shapes only. This vastly simplifies the specialisation process but comes at the price of lack in expressiveness. Only uniform array operations can be defined which immediately rules out the definition of operations such as `take` or `create`.

A similar situation can be found in the C++ based approach to generic array programming called Blitz [Vel98]. There, the rank information is made a template parameter which is resolved statically. Using the template mechanism as a tool for partial evaluation (for details see [Vel99]) results in rank specific C code that — at compile time — is derived from otherwise generic program specifications. This way, similar to the Fish approach, the rank computation is strictly separated from the value computation, as the template mechanism in C++ is strictly separated from the rest of the language.

Further work on specialising generic programs for data types rather than values can be found in the context of algebraic data types (ADT for short). Programs that are defined on generalisations of ADTs as they can be found in the generics of Clean [Ali05], the generic type classes of the Glasgow Haskell Compiler [HP00] or in Generic-Haskell [CHJ+01], when left unspecialised, lead

to significant runtime overhead [AS04]. To ameliorate that problem, Alimarine and Smetsers in [AS04] propose specialisation to data types throughout generic programs. They show that for non-recursive data types this specialisation can be done always without risking non-termination which suggests a brute-force approach similar to online partial evaluation. Although this is similar to the specialisation approach in generic array programming there is a major difference to be observed: in CLEAN, the underlying type system precludes types to depend on argument values. As a consequence, generic array programming that would allow definitions of functions such as `take` or `create` can only be done, if array shapes are part of the data itself. In that case specialisation beyond the level of data types would be required which is outside the scope of the work described in [AS04].

9 Conclusions

This paper proposes an inference algorithm for analysing the relation between the shapes of arguments and the shapes of return values of function definitions in a first order functional array language. It determines for each function which level of argument shape knowledge is required in order to determine a certain level of return shape knowledge. This information can be used to steer function specialisation in a way that ensures that all shapes are computed statically, whenever possible. Once all functions are specialised to appropriate level, the provided shape information can be utilised for various optimisations that are essential for achieving highly efficient runtime behaviour.

With this apparatus at hand, abstractions can be chosen freely without preventing the compiler from applying sophisticated optimisations that are restricted to the intra-procedural case. As a consequence, non-uniform functions such as `take` can be used as building blocks for large applications without introducing considerable runtime degradation.

References

[Ali05] A. Alimarine. *Generic Functional Programming.* PhD thesis, Radboud University of Nijmegen, Netherlands, 2005.

[AS04] A. Alimarine and S. Smetsers. Optimizing Generic Functions. In D. Kozen, editor, *The 7th International Conference, Mathematics of Program Construction, Stirling, Scotland, UK*, volume 3125 of *LNCS*, pages 16–31. Springer, 2004.

[Can89] D.C. Cann. Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1989.

[CHJ+01] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Witt. *The Generic Haskell User's Guide*, 2001.

[GS99] C. Grelck and S.B. Scholz. Accelerating APL Programs with SAC. *SIGAPL Quote Quad*, 29(2):50–58, 1999.

[GT04] C. Grelck and K. Trojahner. Implicit Memory Management for SaC. In C. Grelck and F. Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, pages 335–348. University of Kiel, 2004.

[HP00] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 4th Haskell Workshop*, 2000.

[Hui95] R. Hui. Rank and Uniformity. *APL Quote Quad*, 25(4):83–90, 1995.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[JMB98] C.B. Jay, E. Moggi, and G. Bellè. Functors, Types and Shapes. In R. Backhouse and T. Sheard, editors, *Workshop on Generic Programming: Marstrand, Sweden, 18th June, 1998*, pages 21–4. Chalmers University of Technology, 1998.

[Jon96] N.D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3), 1996.

[JS98] C.B. Jay and P.A. Steckler. The Functional Imperative: Shape! In Chris Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98 Held as part of the joint european conferences on theory and practice of software, ETAPS'98 Lisbon, Portugal, March/April 1998*, volume 1381 of *LNCS*, pages 139–53. Springer-Verlag, 1998.

[Kre03] D.J. Kreye. *A Compiler Backend for Generic Programming with Arrays*. PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2003.

[LLS98] E.C. Lewis, C. Lin, and L. Snyder. The Implementation and Evaluation of Fusion and Contraction in Array Languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*. ACM, 1998.

[Pie02] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 0-262-16209-1.

[Sch03] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[Vel98] T.L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, LNCS. Springer, 1998.

[Vel99] T.L. Veldhuizen. C++ Templates as Partial Evaluation. In O. Danvy, editor, *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18. University of Aarhus, Dept. of Computer Science, 1999.