



## UvA-DARE (Digital Academic Repository)

### Dependently typed array programs don't go wrong

Trojahner, K.; Grelck, C.

**DOI**

[10.1016/j.jlap.2009.03.002](https://doi.org/10.1016/j.jlap.2009.03.002)

**Publication date**

2009

**Document Version**

Final published version

**Published in**

The Journal of Logic and Algebraic Programming

[Link to publication](#)

**Citation for published version (APA):**

Trojahner, K., & Grelck, C. (2009). Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming*, 78(7), 643-664.  
<https://doi.org/10.1016/j.jlap.2009.03.002>

**General rights**

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

**Disclaimer/Complaints regulations**

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.



Contents lists available at ScienceDirect

# The Journal of Logic and Algebraic Programming

journal homepage: [www.elsevier.com/locate/jlap](http://www.elsevier.com/locate/jlap)

## Independently typed array programs don't go wrong

Kai Trojahnner\*, Clemens Grelck

Institute of Software Technology and Programming Languages, University of Lübeck, Germany

### ARTICLE INFO

#### Article history:

Available online 8 April 2009

#### Keywords:

Array programming  
Dependent types  
Program verification

### ABSTRACT

The array programming paradigm adopts multidimensional arrays as the fundamental data structures of computation. Array operations process entire arrays instead of just single elements. This makes array programs highly expressive and introduces data parallelism in a natural way. Array programming imposes non-trivial structural constraints on ranks, shapes, and element values of arrays. A prominent example where such constraints are violated are out-of-bound array accesses. Usually, such constraints are enforced by means of run time checks. Both the run time overhead inflicted by dynamic constraint checking and the uncertainty of proper program evaluation are undesirable.

We propose a novel type system for array programs based on dependent types. Our type system makes dynamic constraint checks obsolete and guarantees orderly evaluation of well-typed programs. We employ integer vectors of statically unknown length to index array types. We also show how constraints on these vectors are resolved using a suitable reduction to integer scalars. Our presentation is based on a functional array calculus that captures the essence of the paradigm without the legacy and obfuscation of a fully-fledged array programming language.

© 2009 Elsevier Inc. All rights reserved.

### 1. Introduction

In the array programming paradigm multidimensional arrays serve as the fundamental data structures of computation. Such arrays can be vectors, matrices, tensors, or structures with an even higher number of axes. Scalar values, such as integer numbers or characters, form the important special case of arrays with zero axes. Array operations work on entire arrays rather than individual elements. This makes array programs highly expressive and introduces data parallelism in a natural way. Hence, functional array programs lend themselves well for parallel execution on parallel computers such as recent multi-core processors [1,2]. Prominent examples of array languages are APL [3], J [4], MATLAB [5], and SAC [6].

A powerful concept found in array programming languages is shape-generic programming: Individual operations and entire algorithms can be specified for arrays of arbitrary size and even an arbitrary number of axes. For example, element-wise arithmetic works for scalars as well as for vectors and matrices. However, this flexibility introduces some non-trivial constraints between function arguments. Element-wise addition requires both arguments to have the same number of axes and the same number of elements along each axis. The constraints are more complicated for operations like array access: the selection of an array element requires the length of the vector of indices to match the number of axes of the array to select from. Moreover, all elements of the index vector must range within the index bounds of the array.

Interpreted array languages like APL, J, and MATLAB are dynamically typed. They feature a large number of built-in operations that implicitly perform the necessary consistency checks on the structural properties of their arguments on each application. In contrast, SAC is a compiled language aimed at high run time performance and automatic parallelization [2]. SAC has a static type system that employs three layers of array types. While the array element type is always monomorphic,

\* Corresponding author.

E-mail addresses: [trojahnner@isp.uni-luebeck.de](mailto:trojahnner@isp.uni-luebeck.de) (K. Trojahnner), [grelck@isp.uni-luebeck.de](mailto:grelck@isp.uni-luebeck.de) (C. Grelck).

structural array properties can be described at three different levels of accuracy: complete information on number of axes and extents, partial information on number of axes but not their extents, and no structural information at all. Using types with complete structural information allows the compiler to statically resolve certain classes of structural constraints. However, complete specification of all array types runs counter the software engineering desire for generic and abstract specifications and code reuse. Code specialization [7] and partial evaluation techniques [8] address this problem, but their success is program dependent. In general, dynamic consistency checks remain prevalent in compiled code. For a language like SAC this is particularly undesirable because run time checks cause overhead both directly through their mere execution and indirectly by hampering program optimization.

In either setting, interpreted or compiled, dynamic consistency checks have a further disadvantage beyond performance considerations: a program may abort with an error message at any given time. In particular, for long-running or safety-critical applications such run time errors are undesirable.

In our current work, we aim at verifying array programs entirely statically. All structural constraints are enforced at compile time by means of a novel type system that combines subtyping with a variant of indexed types [9,10]. Terms denoting integer vectors are used to index an array type of a particular shape from the family of array types. As the length of a shape vector varies with the number of array axes, the sort of the index vector itself is indexed from a sort family using an integer. For example, the type of element-wise addition of integer arrays concisely expresses the required equality on argument and result shapes:

$$\text{add} : \Pi d :: \text{nat}. \Pi s :: \text{natvec}(d). [\text{int}|s] \rightarrow [\text{int}|s] \rightarrow [\text{int}|s]$$

Our type system rules out applications of the function `add` for which the arguments cannot be proved to have equal shape. Thus, program execution can take place without any run time checks. Furthermore, the structural information provided by these array types allow a compiler to perform extensive program optimization. For specific arrays, singleton types even capture the value of an array's elements. Similar to other approaches based on indexed types such as DML [11], type checking proceeds by checking constraints on linear integer expressions. In the system presented in this paper, all well-typed programs are guaranteed not to exhibit any undesired behavior at run time. A particular challenge in our context is to efficiently resolve constraints between integer vectors of statically unknown length.

Our approach is rather disruptive than incremental for any existing array programming language. Hence, we first develop our type system for an abstract functional array calculus that captures the essence of array programming without the legacy problems of a fully-fledged programming language. We follow the example of SAC, but leave out all aspects irrelevant to our work (e.g. the module and state systems) and somewhat streamline the remaining parts. Our calculus has some features currently not supported by SAC, e.g. higher-order functions and non-homogeneous nestings of multidimensional arrays.

We make the following contributions:

- We specify a language with the essential features necessary for shape-generic functional array programming with dependent types that allows for both higher-order functions and complex nestings of multidimensional arrays.
- We define a type system for the static verification of dependently typed array programs that combines subtyping with a novel variant of indexed types that uses integer vectors of statically unknown length to index elements of type families.
- We propose a scheme for mapping the resolution of constraints on integer vectors of arbitrary length to linear integer constraints that may be processed by standard SMT solvers.

Our approach provides a solution for type-safe functional array programming: any well-typed array program is guaranteed to yield a proper value. In short: Dependently typed array programs don't go wrong!

The paper is organized as follows: Section 2 gives a gentle introduction to multidimensional arrays. In Section 3 we introduce our calculus for functional array programming and present its small-step semantics. Section 4, illustrates the kind of programs we are interested in and motivates our type system for the static verification of array programs described in Section 5. We outline our concept for vector constraint resolution in Section 6. Finally, we discuss related work in Section 7 and draw conclusions in Section 8.

## 2. Multidimensional arrays

A characteristic feature of array programming languages is that only arrays are values, i.e. legitimate results of computations. Arrays may be vectors, matrices, tensors, or structures with an even higher number of axes. In particular, arrays may also be scalar values (such as the integers) which form the important special case of arrays without any axes. The appropriate abstraction which allows for treating different kinds of arrays in a uniform way are truly multidimensional arrays.

Multidimensional arrays are characterized by two essential properties: a scalar rank and a shape vector. The *rank* denotes an array's number of axes. Its *shape vector* contains the array's extent along each axis. For a given array, the length of its shape vector equals its rank. Fig. 1 shows a few examples of multidimensional arrays and their basic properties. The scalar array 1 does not have any axes and hence its shape vector is empty. Vectors have a single axis, so the shape vector of  $[1\ 2\ 3]$  is  $[3]$ . The scheme extends to arrays with an arbitrary number of axes.

The shape vector determines the number of elements in an array. Let  $A$  be an array of rank  $r$  and shape vector  $s$ . Then the number of elements in  $A$  is given by the equation

$$|A| = \prod_{i=1}^r s_i.$$

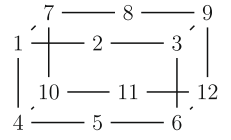
Array	Rank	Shape vector
1	0	$[]$
$[1\ 2\ 3]$	1	$[3]$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	2	$[2\ 3]$
	3	$[2\ 2\ 3]$

Fig. 1. Ranks and shape vectors of the example arrays.

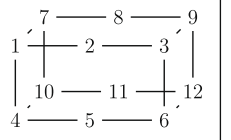
Array	Index vectors
1	$[]$
$[1\ 2\ 3]$	$[[0]\ [1]\ [2]]$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$\begin{pmatrix} [0\ 0] & [0\ 1] & [0\ 2] \\ [1\ 0] & [1\ 1] & [1\ 2] \end{pmatrix}$
	$\begin{matrix} [1\ 0\ 0] - [1\ 0\ 1] - [1\ 0\ 2] \\ [0\ 0\ 0] - [0\ 0\ 1] - [0\ 0\ 2] \\ [1\ 1\ 0] - [1\ 1\ 1] - [1\ 1\ 2] \\ [0\ 1\ 0] - [0\ 1\ 1] - [0\ 1\ 2] \end{matrix}$

Fig. 2. Example arrays and the legal index vectors.

Individual elements are selected from an array with  $n$  axes by means of an *index vector* of length  $n$ . Both the index vector and the selected element are arrays themselves. Fig. 2 gives an overview of the admissible index vectors into the arrays from Fig. 1. The first row again shows the special case of scalar arrays: as the array 1 does not have any axes, the empty vector is the only legal index vector. Such a selection again yields the array 1. The other cases are more straightforward. For example, we may index into a matrix using appropriate index vectors of length two.

A more rigorous syntax for multidimensional arrays is shown in Fig. 3 along with a suitable evaluation relation for evaluating array terms. We use the notation  $a^n$  to represent comma separated lists  $a_1, \dots, a_n$ . In order to express that a property holds for all elements of a sequence  $a^n$  we write  $\forall i. p(a_i)$  instead of  $\forall i. 1 \leq i \leq n \Rightarrow p(a_i)$ . Array values have the form  $[q^p \mid s^d]$ . In such an array, the integer vector  $s^d$  represents the shape vector; its length  $d$  encodes the array's rank. The *data vector*  $q^p$  contains the array elements as a sequence of *quarks*. For the moment, quarks may only be integers but we will introduce other kinds of quarks in Section 3. Quarks owe their name to the fact that array programs employ arrays as the atomic units of computation (all values in the system are arrays). Hence, array elements must be a subatomic particles.

Fig. 4 shows the array values corresponding to the example arrays. We demand that array values adhere to a data type invariant:  $[q^p \mid s^d]$  is valid iff no axis has negative length and the number of quarks equals the product of the elements of the shape vector:

- (1)  $\forall i. s_i \geq 0$ ,
- (2)  $p = \prod_{i=1}^d s_i$ .

Inside the data vector, the elements along the innermost array axis are stored consecutively (row-major order). For multidimensional arrays, this means that the order of elements is determined by the lexicographic order of the corresponding index vectors. Let  $A$  be an array of rank  $r$  and shape  $s$ , and let  $v$  be a suitable index vector for  $A$ . The function  $\iota$  then determines the linear index of the element at position  $v$  in the data vector of  $A$ :

$$\iota(r, s^r, v^r) = \sum_{i=1}^r (v_i \cdot \prod_{j=i+1}^r s_j) + 1.$$

Properties of arrays can be accessed using three primitives: `rank`, `shape`, and `se1`. All operations first evaluate their arguments to array values and then yield an array containing the desired properties themselves. For an array  $A = [q^p \mid s^d]$ , `rank A` evaluates to the integer scalar  $d$ , represented as  $[d \mid []]$ . The term `shape A` yields the shape vector of  $A$  in the form  $[s^d \mid [d]]$ . As an example, we apply both functions to a matrix of shape  $2 \times 3$ :

$$\text{rank } [1, 2, 3, 4, 5, 6 \mid [2, 3]] \longrightarrow [2 \mid []]$$

$$\text{shape } [1, 2, 3, 4, 5, 6 \mid [2, 3]] \longrightarrow [[2, 3] \mid [2]]$$

Syntactic forms

$t ::= [q^p \mid [s^d] \mid ] \mid \text{rank } t \mid \text{shape } t \mid \text{sel}(t, t)$	Terms
$q ::= c$	Quarks
$v ::= [q^p \mid [s^d] \mid ]$	Values

Evaluation rules

$\frac{t \longrightarrow t'}{\text{rank } t \longrightarrow \text{rank } t'}$	$\text{rank } [q^p \mid [s^d] \mid ] \longrightarrow [d \mid [ ] \mid ]$
$\frac{t \longrightarrow t'}{\text{shape } t \longrightarrow \text{shape } t'}$	$\text{shape } [q^p \mid [s^d] \mid ] \longrightarrow [s^d \mid [d] \mid ]$
$\frac{t_1 \longrightarrow t'_1}{\text{sel}(t_1, t_2) \longrightarrow \text{sel}(t'_1, t_2)}$	$\frac{t_2 \longrightarrow t'_2}{\text{sel}(v_1, t_2) \longrightarrow \text{sel}(v_1, t'_2)}$
$\forall k. 0 \leq i_k < s_k$	
$\frac{}{\text{sel}([q^p \mid [s^d] \mid ], [i^d \mid [d] \mid ]) \longrightarrow [q_{\iota(d, s^d, i^d)} \mid [ ] \mid ]}$	

Fig. 3. A core system for representing and accessing multidimensional arrays.

Array	Uniform array representation
1	$[1 \mid [ ] \mid ]$
1 2 3	$[1, 2, 3 \mid [3] \mid ]$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$[1, 2, 3, 4, 5, 6 \mid [2, 3] \mid ]$
$\begin{array}{ccc} 7 & \text{---} & 8 & \text{---} & 9 \\ 1 & \text{---} & 2 & \text{---} & 3 \\ \vdots & & \vdots & & \vdots \\ 10 & \text{---} & 11 & \text{---} & 12 \\ 4 & \text{---} & 5 & \text{---} & 6 \end{array}$	$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \mid [2, 2, 3] \mid ]$

Fig. 4. Uniform representations of the example arrays.

Since the application of `shape` to an array results in a vector whose length equals the given array's rank, one may think that applying `shape` twice is another way to obtain the rank, making the `rank` primitive obsolete. However, the results are not the same because `shape` will always evaluate to a vector whereas `rank` yields a scalar.

$$\text{shape}(\text{shape } [q^n \mid [s^d] \mid ]) \longrightarrow^* [d \mid [1] \mid ]$$

$$\text{rank } [q^n \mid [s^d] \mid ] \longrightarrow [d \mid [ ] \mid ]$$

A selection  $\text{sel}(A, [i^e \mid [e] \mid ])$  into a multidimensional array  $A = [q^p \mid [s^d] \mid ]$  is evaluated if two constraints are met. Firstly, the length  $e$  of the index vector must equal the rank  $d$  of  $A$ . Secondly, the index vector  $i^e$  must actually denote a valid position in  $A$ , i.e. the values of all quarks  $i_k$  must range between 0 and  $s_k$ . The selection will then evaluate to a scalar array whose sole quark is taken from the data vector of  $A$  at position  $\iota(d, s^d, i^d)$ .

Selections with index vectors of invalid length or index vectors denoting a position outside the array boundaries cannot be evaluated and are thus program errors. To illustrate array selection, we select the central element from a matrix of shape  $[3, 3]$ :

$$\frac{0 \leq 1 < 3 \wedge 0 \leq 1 < 3}{\text{sel}([1, 2, 3, 4, 5, 6, 7, 8, 9 \mid [3, 3] \mid ], [1, 1 \mid [2] \mid ]) \longrightarrow [5 \mid [ ] \mid ]}$$

The evaluation rules for both `rank` and `shape` are straightforward: Whenever the argument reduces to a value, a result will be provided. In contrast, successful evaluation of selections depends on non-trivial constraints between the arguments' ranks, shape vectors, and the values of the array elements.

We have introduced the main ideas of multidimensional arrays with a custom syntax for arrays and a semantics for the essential array operations. In the next section, we will extend these ideas towards a core language for functional array programming. To pinpoint potential program errors, we will provide a detailed small-step semantics for our calculus.

### 3. A core functional array programming language

The core language allows for the type-safe specification of shape-generic array programs. Such programs operate on arrays with an arbitrary shape and even with an arbitrary number of axes. We deliberately leave out several features of

$I$	::=	$\text{idx} \mid \text{idxvec}(i) \mid \{l \text{ in } ir\}$	Index sorts
$i$	::=	$c \mid x \mid [i^n] \mid \bar{f}(i, i) \mid f_2(i, i)$	Index terms
$ir$	::=	$i \mid i.. \mid ..i \mid i..i$	Index ranges
$T$	::=	$[Q \mid i] \mid S(i)$	Types
$Q$	::=	$\perp_Q \mid \text{int} \mid T \rightarrow T \mid \Pi x :: I. T \mid \{T^n\} \mid \Sigma x :: I. T$	Quark types
$S$	::=	$\text{num} \mid \text{numvec}$	Singleton types
$t$	::=	$[[q^p \mid [c^n]]] \mid x \mid t \mid t' \mid i$ $\mid \text{let } x = t \text{ in } t \mid \{t^n\} \mid \text{let } \{x^n\} = t \text{ in } t$ $\mid \{i, t : \Sigma x :: I. T\} \mid \text{let } \{x, x\} = t \text{ in } t$ $\mid [t^p \mid [c^n]] \mid f \ t \mid \text{gen } x < t \text{ of } t \text{ with } t$ $\mid \text{loop } x < t, x = t \text{ with } t \mid \text{case } t \text{ in } m$	Terms
$q$	::=	$c \mid \lambda x : T. t \mid \lambda' x :: I. t \mid \{v^n\} \mid \{i, v : \Sigma x :: I. T\}$	Quarks
$m$	::=	$r \Rightarrow t \mid m \mid \text{else} \Rightarrow t$	Matches
$r$	::=	$t \mid t.. \mid ..t \mid t..t$	Ranges
$f$	::=	$\bar{f} \mid f_2 \mid \text{rank} \mid \text{shape} \mid \text{length} \mid \text{sel}$	Built-ins
$\bar{f}$	::=	$\text{vec} \mid ++ \mid \text{take} \mid \text{drop}$	Vector ops
$f_2$	::=	$+ \mid - \mid \text{min} \mid \text{max}$	Dyadic ops
$v$	::=	$[[q^p \mid [c^n]]]$	Values
$rv$	::=	$v \mid v.. \mid ..v \mid v..v$	Value ranges

Fig. 5. Syntax of a core language for typed functional array programming.

functional programming languages that would unnecessarily complicate the presentation in this paper. Among others, the core language does not support polymorphism, algebraic data types, and general recursion. Nonetheless, since all these features are largely orthogonal to our approach, we are confident they could be soundly integrated.

To rule out program errors such as the invalid array selection the language employs types for arrays that describe both the type of the quarks inside an array as well as its shape. In particular, the shape component of a type is itself an expression. This makes our array types a variant of dependent types. However, as type checking of programs with full dependent types is generally undecidable, we restrict the shape expressions to a dedicated *index language* in which only predefined and well-behaved (i.e. linear) operations are permitted. Type checking then reduces to solving constraints over these index terms.

The syntax of the language is shown in Fig. 5; its operational semantics is shown in Figs. 6–8. The language description can be divided into three conceptual sections: The top section defines the index language which is used to index types from the type families. The next section describes the types used in the system. The remainder of the figure defines the term language, namely the quarks and array terms. The discussion in this section will follow the same route.

### 3.1. Index language

Types may only depend on the terms of a specific index language. These index terms are solely used for type checking; they are not subject to evaluation. All index terms belong to an index sort.  $\text{idx}$  is the sort of integer scalars,  $\text{idxvec}(i)$  is the sort-family of integer vectors. In this sort family, a sort for vectors of a particular length is designated using a scalar index term  $i$ . We use index vectors to index into the family of multidimensional array types.

Scalar index terms are integer constants  $c$ , variables of sort  $\text{idx}$ , and applications of linear dyadic functions such as addition and subtraction to scalar index terms. Index vectors may also be variables of a vector sort, but can be constructed from scalar index terms as well. For example, the index vector  $[0, 1, 2]$  belongs to the sort  $\text{idxvec}(3)$ . We may also apply binary linear functions to index vectors of equal length. This yields another index vector of that sort by element-wise application of the given function. In particular, we may form vectors whose length is given by a scalar index term. For a non-negative scalar index  $l$  and another scalar index  $i$ ,  $\text{vec}(l, i)$  yields an index vector of length  $l$  whose elements all equal  $i$ . There are also index vector terms that map between the index sorts. Vectors may be concatenated using  $a ++ b$  which appends the vector  $b$  of length  $l_b$  to the vector  $a$  of length  $l_a$ . Naturally, the result is of sort  $\text{idxvec}(l_a + l_b)$ . Conversely, vectors can be split using the operations  $\text{take}$  and  $\text{drop}$ . For a given vector  $v$  of length  $l$  and a scalar index expression  $i$  with  $0 \leq i \leq l$ ,  $\text{take}(i, v)$  and  $\text{drop}(i, v)$  denote the prefix of  $v$  with length  $i$  and the suffix of  $v$  with length  $l - i$ , respectively. Thus we have  $\text{take}(i, v) ++ \text{drop}(i, v) = v$ .

Index sorts can be restricted to specific ranges using the subset notation  $\{l \text{ in } ir\}$ . Given two scalar index terms  $a$  and  $b$ , the sort  $\{\text{idx in } a..b\}$  denotes all  $x$  of sort  $\text{idx}$  for which  $a \leq x < b$ . Both boundaries may be omitted, indicating  $\pm \infty$  as the boundaries. A sort of the form  $\{l \text{ in } i\}$  denotes a sort that contains  $i$  as its single element. In the following we will use  $\text{nat} = \{\text{idx in } 0..\}$  and  $\text{natvec}(i) = \{\text{idxvec}(i) \text{ in } \text{vec}(i, 0)..\}$ .

$$\begin{array}{c}
\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \text{ (E-APP1)} \qquad \frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2} \text{ (E-APP2)} \\
\frac{}{[\lambda x : T. t \mid \square] v_2 \longrightarrow t[x \mapsto v_2]} \text{ (E-APPABS)} \\
\frac{t \longrightarrow t'}{t' i \longrightarrow t' i} \text{ (E-IAAPP)} \\
\frac{}{[\lambda' x :: I. t \mid \square] i \longrightarrow t[x \mapsto i]} \text{ (E-IAPIABS)} \\
\frac{t_j \longrightarrow t'_j}{\{v^{j-1}, t_j, t^{n-j}\} \longrightarrow \{v^{j-1}, t'_j, t^{n-j}\}} \text{ (E-TUP1)} \\
\frac{}{\{v^n\} \longrightarrow [\{v^n\} \mid \square]} \text{ (E-TUP2)} \\
\frac{t \longrightarrow t'}{\{i, t : \Sigma x :: I. T\} \longrightarrow \{i, t' : \Sigma x :: I. T\}} \text{ (E-ITUP1)} \\
\frac{}{\{i, v : \Sigma x :: I. T\} \longrightarrow [\{i, v : \Sigma x :: I. T\} \mid \square]} \text{ (E-ITUP2)} \\
\frac{t_1 \longrightarrow t'_1}{\text{let } p = t_1 \text{ in } t_2 \longrightarrow \text{let } p = t'_1 \text{ in } t_2} \text{ (E-LET)} \\
\frac{}{\text{let } x = v_1 \text{ in } t_2 \longrightarrow t_2[x \mapsto v_1]} \text{ (E-LETVAL)} \\
\text{let } \{x^i\} = [\{v^i\} \mid \square] \text{ in } t_2 \longrightarrow t_2[x_1 \mapsto v_1]..[x_n \mapsto v_n] \text{ (E-LET TUP)} \\
\text{let } \{x_1, x_2\} = [\{i, v : \Sigma x :: I. T\} \mid \square] \text{ in } t_2 \longrightarrow t_2[x_1 \mapsto i][x_2 \mapsto v] \text{ (E-LET ITUP)}
\end{array}$$

Fig. 6. Basic semantics of typed array programs.

### 3.2. Types for array programs

There are two major kinds of types for array programs: quark types for describing the quarks inside an array and array types for describing entire arrays through its quark type and its shape. Quark types and array types follow the mutually recursive structure of quarks and array values. The array type  $[Q \mid i]$  describes all arrays whose elements have quark type  $Q$  and whose shape vector is characterized by the index vector  $i$ . For example, the type of an integer vector  $[\text{int} \mid [4]]$ , while a scalar integer  $[\text{int} \mid \square]$  has type  $[\text{int} \mid \square]$ .

The integer quarks of type  $\text{int}$  are the only primitive values used in the language. Clearly, other base types could be supported as well. In addition, there are also structured quarks: abstractions  $\lambda x : T_1. t$  of type  $T_1 \rightarrow T_2$ , index abstractions  $\lambda' x :: I. t$  of type  $\Pi x :: I. T$ , tuples of arrays values  $\{v_1, \dots, v_n\}$  of type  $\{T_1, \dots, T_n\}$ , and dependent pairs  $\{i, v : \Sigma x :: I. T\}$  of type  $\Sigma x :: I. T$ . The bottom quark type  $\perp_Q$  is not associated with a particular quark. Instead, it serves as a quark type for empty arrays such as the empty vector  $[\square \mid [0]]$  which has type  $[\perp_Q \mid [0]]$ . To capture the intuition that an empty array is compatible with arrays of any quark type,  $\perp_Q$  is a subtype of every quark type.

Due to the significance of integer scalars and vectors for array programs, we provide singleton types for these arrays that do not only characterize their shape, but also the values of the contained integer quarks. The type  $\text{num}(i)$  characterizes all scalar integer arrays whose quark is identical to the index  $i$ . By means of subtyping, each  $\text{num}(i)$  is also an  $[\text{int} \mid \square]$ . Similarly, an integer vector of type  $\text{numvec}(i)$  is also an  $[\text{int} \mid [i]]$  provided that the index vector  $i$  is of sort  $\text{idvec}(I)$ . Thus, the above arrays  $[\text{int} \mid \square]$  and  $[\text{int} \mid [4]]$  also have the more specific types  $\text{num}(7)$  and  $\text{numvec}([1, 2, 3, 4])$ , respectively.

### 3.3. Syntax and semantics of array programs

We now explain the syntax and semantics of the terms of the array language. The evaluation rules of the basic language elements are defined in Fig. 6.

**Functions.** The abstraction quark  $\lambda x : T_1. t$  allows us to specify arrays of functions. Its type is the function quark type  $T_1 \rightarrow T_2$ . The application  $t_1 t_2$  is explained by the evaluation rules E-APP1, E-APP2, and E-APPABS. Following a call-by-value regime, the application first evaluates both the operator  $t_1$  and the operand  $t_2$ . Only if  $t_1$  evaluates to a scalar array with a single abstraction  $[\lambda x : T. t \mid \square]$ , the entire application will take a  $\beta$ -reduction step by substituting all free occurrences of  $x$  in  $t$  with the evaluated argument.

The index abstraction quark  $\lambda' x :: I. t$  allows us to abstract an index variable from both terms and types. The type of the index abstraction is  $\Pi x :: I. T$ , where  $T$  may refer to the index identifier  $x$ . By abstracting an index vector from the shape

$$\begin{array}{c}
\frac{t \longrightarrow t'}{f t \longrightarrow f t'} \text{ (E-PRFAPP)} \\
\text{rank } [q^p \mid [s^d]] \longrightarrow [d \mid []] \text{ (E-RANK)} \\
\text{shape } [q^p \mid [s^d]] \longrightarrow [s^d \mid [d]] \text{ (E-SHAPE)} \\
\text{length } [q^l \mid [l]] \longrightarrow [l \mid []] \text{ (E-LENGTH)} \\
f_2 [ [q^p \mid [s^d]] , [r^p \mid [s^d]] ] \longrightarrow [f_2(q_1, r_1), \dots, f_2(q_p, r_p) \mid [s^d]] \text{ (E-BIN)} \\
\\
\frac{\forall k. 0 \leq i_k < s_i}{\text{sel } [ [ [q^p \mid [s^d]] , [i^d \mid [d]] ] ] \longrightarrow [q_{i(d, s^d, i^d)} \mid []] ]} \text{ (E-SEL)} \\
\\
\frac{l \geq 0}{\text{vec } [ [ [l \mid []] , [q \mid []] ] ] \longrightarrow [ \underbrace{q, \dots, q}_l \mid [l] ] ]} \text{ (E-VEC)} \\
\\
++ [ [ [q^m \mid [m]] , [q^n \mid [n]] ] ] \longrightarrow [q^m, q^n \mid [m+n]] \text{ (E-CAT)} \\
\\
\frac{0 \leq n \leq l}{\text{take } [ [ [n \mid []] , [q^l \mid [l]] ] ] \longrightarrow [q_1, \dots, q_n \mid [n]] ]} \text{ (E-TAKE)} \\
\\
\frac{0 \leq n \leq l}{\text{drop } [ [ [n \mid []] , [q^l \mid [l]] ] ] \longrightarrow [q_{n+1}, \dots, q_l \mid [l-n]] ]} \text{ (E-DROP)} \\
\\
\frac{t_j \longrightarrow t'_j}{[t^{j-1}, t_j, t^{n-j} \mid [f^d]] \longrightarrow [t^{j-1}, t'_j, t^{n-j} \mid [f^d]]} \text{ (E-ARR1)} \\
\\
[q_i^p \mid [c^e]]^n \mid [f^d] \longrightarrow [q_i^p, \dots, q_n^p \mid [f^d, c^e]] \text{ (E-ARR2)} \\
\\
\frac{t_1 \longrightarrow t'_1}{\text{gen } x < t_1 \text{ of } t_2 \text{ with } t_3 \longrightarrow \text{gen } x < t'_1 \text{ of } t_2 \text{ with } t_3} \text{ (E-GENF)} \\
\\
\frac{t_2 \longrightarrow t'_2}{\text{gen } x < v_1 \text{ of } t_2 \text{ with } t_3 \longrightarrow \text{gen } x < v_1 \text{ of } t'_2 \text{ with } t_3} \text{ (E-GENC)} \\
\\
\frac{\forall k. f_k \geq 0 \quad \exists j. f_j = 0}{\text{gen } x < [f^d \mid [d]] \text{ of } [c^e \mid [e]] \text{ with } t \longrightarrow [ [f^d, c^e] ]} \text{ (E-GENE)} \\
\\
\frac{\forall k. f_k > 0 \quad \forall y^d \in \vec{0}. f^d. c_{i(d, f^d, y^d)} = t[x \mapsto_i [y^d]] [x \mapsto [y^d \mid [d]]]}{\text{gen } x < [f^d \mid [d]] \text{ of } v \text{ with } t \longrightarrow [c^p \mid [f^d]]} \text{ (E-GEN)} \\
\\
\frac{t_1 \longrightarrow t'_1}{\text{loop } x_1 < t_1, x_2 = t_2 \text{ with } t_3 \longrightarrow \text{loop } x_1 < t'_1, x_2 = t_2 \text{ with } t_3} \text{ (E-LOOP1)} \\
\\
\frac{\forall k. 0 \leq s_k \quad \forall v^d \in \vec{0}. f_{i(d, s^d, v^d)} = [\lambda y. t_3[x \mapsto_i [v^d]]] [x \mapsto [v^d \mid [d]]] \mid [] \mid []}{\text{loop } x < [s^d \mid [d]] , y = t_2 \text{ with } t_3 \longrightarrow f_p \dots (f_1 t_2)} \text{ (E-LOOP2)}
\end{array}$$

Fig. 7. Semantics of the array specific built-in operations.

of a function argument, we can specify operations applicable to arrays of arbitrary shape. Taking this idea further, we may abstract the length from this index argument and obtain a rank-generic function.

Index abstractions are applied to index arguments with the index application  $t \ 'i$ . As defined by the evaluation rules E-IAPP and E-IAPPIABS, the index application  $t \ 'i$  only evaluates the applied term  $t$  but not the index argument  $i$ . Provided that  $t$  evaluates to a scalar array with a single index abstraction quark  $[\lambda'x :: I. t \mid []]$ , the index application takes an evaluation step by substituting all index identifiers  $x$  in  $t$  with  $i$ .

*Tuples.* Besides constants and (dependent) functions, arrays may also contain  $n$ -ary tuples of arrays and dependent pairs that couple index terms with arrays. The tuple quark  $\{v_1, \dots, v_n\}$  of type  $\{T_1, \dots, T_n\}$  encloses  $n$  array values into a single quark, thus allowing for arrays containing (tuples of) arrays.

Since all quarks in an array must have a common type, tuples only allow for uniform nestings in which all inner arrays have the same shape. This restriction is overcome with the dependent pair quark  $\{i, v : \Sigma x :: I. T\}$  of type  $\Sigma x :: I. T$ . In a dependent



pair, the type of the second component may depend on the index that is the first component. The type annotation  $\Sigma x :: I. T$  is necessary because the typing of a dependent pair is ambiguous. For example, the dependent pair  $\{2, [12, 2 | [2] []]\}$  has type  $\Sigma x :: \text{nat}. [\text{int} | [x]]$ , but also the types  $\Sigma x :: \text{nat}. [\text{int} | [2]]$ ,  $\Sigma x :: \text{nat}. \text{numvec}([x, x])$ , and  $\Sigma x :: \text{nat}. \text{numvec}([2, x])$ , among others. Vice versa, several dependent pairs have the same type: both dependent tuples  $\{2, [12, 2 | [2] []]\}$  and  $\{3, [1, 2, 3 | [3] []]\}$  have the type  $\Sigma x :: \text{nat}. [\text{int} | [x]]$ . Thus, by abstracting a variable from the shapes of the arrays in a dependent pair, we may form nestings of heterogeneous arrays.

Tuple quarks and dependent pair quarks only contain fully evaluated array values. The tuple constructor  $\{t_1, \dots, t_n\}$  is a term that allows to form tuples from arbitrary expressions. It first evaluates all terms  $t_i$  to values  $v_i$  from left to right (E-TUP1) and then reduces to a scalar array with a single tuple quark  $\{v_1, \dots, v_n\}$  according to rule E-TUP2. Analogously, there is also constructor term for dependent pairs  $\{i, t : \Sigma x :: I. T\}$  which is explained by the rules E-ITUP1 and E-ITUP2.

*Let binding.* The `let` binding associates names with the values of complex subterms. As outlined by the evaluation rules E-LET and E-LETVAL, `let  $x = t_1$  in  $t_2$`  first evaluates  $t_1$  to a value and then replaces all free identifiers  $x$  in  $t_2$  with the result. Moreover, the `let` binding serves to unpack tuples and dependent pairs (E-LETTUP, E-LETITUP). Provided that  $t_1$  evaluates to a scalar array with a single tuple quark  $\{v_1, \dots, v_n\}$ , the binding `let  $\{x_1, \dots, x_n\} = t_1$  in  $t_2$`  will evaluate to  $t_2$  in which each identifier  $x_i$  has been replaced with the  $i$ th tuple component  $v_i$  from left to right. Similarly, when  $t_1$  yields a dependent pair  $\{i, v : \Sigma x :: I. T\}$ , `let  $\{x_1, x_2\} = t_1$  in  $t_2$`  will first substitute  $x_1$  with the index term  $i$  in  $t_2$  and then replace  $x_2$  with the value  $v$  in the body.

*Built-in operations.* The operational semantics of the array specific language elements is shown in Fig. 7. The primitives `rank` and `shape` are already known from Section 2. An additional primitive `length` determines the length of a given vector. The operations `+`, `-`, `max`, and `min` can be applied to pairs of shape-conforming integer arrays. Their evaluation is defined by the rule E-BIN as per-element applications of the respective operation. The selection `sel  $\{a, x\}$` , also written  `$a.[x]$` , selects for any valid selection vector  $x$  an element from  $a$ . For any non negative integer  $l$  and scalar array  $b$ , `vec  $\{l, b\}$`  yields a vector of length  $l$  whose elements are all  $b$ . For a vector  $v$  of length  $l$  and an integer  $n$  with  $0 \leq n \leq l$ , `take  $\{l, v\}$`  and `drop  $\{l, v\}$`  yield the prefix of  $v$  with length  $l$  and the suffix of  $v$  with length  $n - l$ , respectively.

*Array construction.* The array constructor  `$[t^n | [f^d]]$`  with  $\forall i. f_i > 0$  and  $n = \prod_{i=1}^d f_i$  creates an array by evaluating the *cell terms*  $t_j$ , which must all evaluate to array values of the same shape. The shape of the newly formed array is prefixed with the *frame shape*  $f^d$ . Its suffix is the common shape vector of the evaluated cells. As shown in the evaluation rules E-ARR1, the cells are evaluated in no specific order, thus introducing a data parallel flavor of concurrency. The data vector of the new array is obtained by concatenating the cells' individual data vectors, e.g.

$$[[1, 2, 3 | [3] []], [4, 5, 6 | [3] []] | [2]] \longrightarrow^* [1, 2, 3, 4, 5, 6 | [2, 3] []].$$

Whereas array constructors statically fix the frame shape, `with`-loops allow for shape-generic array definitions. The concept of the `with`-loop originates from SAC. We have simplified its syntax and semantics for the context of this work. An expression `gen  $x < t_1$  of  $t_2$  with  $t_3$`  defines an array with a frame of shape  $t_1$  that contains cells of the *cell shape*  $t_2$ . Each cell is computed by evaluating the *cell term*  $t_3$  in which  $x$  is assigned the cell's position inside the frame.

Using a `with`-loop, we can for example apply a function  $f$  to each element of an array  $a$ , yielding an array of results:

$$\text{gen } x < \text{shape } a \text{ of } [ [] [0] [] ] \text{ with } (f \ a. [x])$$

Both the frame shape and the cell shape are evaluated before the actual evaluation of the `with`-loop takes place (E-GENF, E-GEND). Provided that  $t_1$  evaluates to a strictly positive integer vector  $[s^d | [d] []]$ , the cell shape may be ignored and the entire expression is evaluated according to rule E-GEN. The `with`-loop evaluates in one step to an array constructor  `$[t_c^p | [s^d]]$` , that in turn evaluates to the result array by the rules E-ARR1 and E-ARR2. Each cell expression  $t_c^p$  is obtained by first substituting the index identifier  $x$  in  $t_3$  with an index vector denoting the cell's position inside the frame and subsequently replacing the regular identifier  $x$  in  $t_3$  with an array of the same content. If  $t_1$  specifies an empty frame shape, the `with`-loop evaluates to an empty array of shape  $t_1 \leftrightarrow t_2$  as stated by rule E-GENE. Having no quarks, the empty array has quark type  $\perp_0$  and is thus compatible with any other quark type.

*Reduction.* The `loop` expression traverses an index space in lexicographic order with a single loop-carried dependency. It is possible to define loops with both scalar and vector boundaries. We restrict our presentation to the latter. In a term of the form `loop  $x_1 < t_1, x_2 = t_2$  with  $t_3$` , the non-negative integer vector  $t_1$  defines the index space.  $t_2$  serves as the *initial value* of the accumulator  $x_2$ . The *loop body*  $t_3$  is evaluated for all non-negative vectors up to  $t_1$  in ascending lexicographic order. Thereby, the current position is bound to the identifier  $x_1$ . The *accumulator*  $x_2$  represents the intermediate loop result. As an example, we provide a loop that computes the sum of integers from an array  $a$  of any shape:

$$\text{loop } x < \text{shape } a, s = [0 | []] \text{ with } s + a.[x]$$

*Conditional.* Finally, the language provides support for a generalized form of a conditional. Its semantics is shown in Fig. 8. The expression `case  $t$  in  $m$`  evaluates to one of multiple branches in  $m$  depending on the value of the integer (vector)  $t$ . The branching condition is first evaluated to a value. This value is then successively compared with the ranges specified in the

$$\begin{array}{c}
\frac{t \longrightarrow t'}{\text{case } t \text{ in } m \longrightarrow \text{case } t' \text{ in } m} \text{ (E-CASE)} \\
\text{case } v \text{ in else} \Rightarrow t \longrightarrow t \text{ (E-ELSE)} \\
\frac{r \longrightarrow r'}{\text{case } v \text{ in } r \Rightarrow t \mid m \longrightarrow \text{case } v \text{ in } r' \Rightarrow t \mid m} \text{ (E-RANGE)} \\
\frac{M(v, rv)}{\text{case } v \text{ in } rv \Rightarrow t \mid m \longrightarrow t} \text{ (E-MATCH)} \\
\frac{\neg M(v, rv)}{\text{case } v \text{ in } rv \Rightarrow t \mid m \longrightarrow \text{case } v \text{ in } m} \text{ (E-NEXT)}
\end{array}$$

Fig. 8. Semantics for conditional expressions.

branches of the form  $r \Rightarrow t \mid m$ . If the value of  $t$  lies in the range  $r$ , the conditional evaluates to  $t$ . Otherwise, the next branch in  $m$  is tried. In case there is no matching branch, the terminal `else`  $\Rightarrow t$  branch will be evaluated.

Using the `case` construct, we may for example define a dynamic check to verify that a selection vector  $x$  points to a valid position in an array  $a$ . In particular, the type checker will make use of this knowledge when it checks the selection  $a.[x]$ :

```
case x in vec {length x, 0} . . shape a  $\Rightarrow$  a.[x] | else  $\Rightarrow$  0
```

In this section, we have presented a core language for type-safe functional array programming. The emphasis lies on the combination of shape-generic programming and dependent types.

#### 4. Shape-generic array programming with dependent types

We now illustrate shape-generic array programming with dependent types with a series of practical examples. To improve legibility, we will employ some notational simplifications. The type of a scalar array is denoted by its quark type  $Q$  instead of its full array type  $[Q \mid []]$ . Similarly, we abbreviate a scalar array value  $[q \mid []]$  with its sole quark  $q$ . To aid the definition of more complex functions, we will use a notation similar to HASKELL programs in which the type declaration and the definition of a function appear on separate lines. The transformation of the notational extensions into the core language is straightforward.

##### 4.1. Shape-generic array operations

The `WITH`-loop allows us to specify shape-generic algorithms. As a first example, we develop a shape-generic `map` operation that applies a function to each element of an array. `map` is a *uniform array operation*, i.e. an operation whose result shape depends solely on the shapes of its arguments. We start with a shape-specific implementation for  $2 \times 2$  matrices:

```
map : (int  $\rightarrow$  int)  $\rightarrow$  [int | [2, 2]]  $\rightarrow$  [int | [2, 2]]
map f a = gen x < [2, 2 | [2] |] of [ | [0] |] with f a.[x]
```

Using dependent types, we can generalize `map` such that it becomes applicable to arbitrary matrices. We abstract the index variable  $s$  from the shape component of the array type. In the definition, we replace the concrete frame shape with `shape a` that gives us the appropriate value. Despite the function's generality, the type states precisely the necessary conformance of the argument and the result shape:

```
map :  $\Pi s :: \text{natvec}(2).$ 
      (int  $\rightarrow$  int)  $\rightarrow$  [int | s]  $\rightarrow$  [int | s]
map 's f a = gen x < shape a of [ | [0] |] with f a.[x]
```

Even more general, by abstracting from the length of the index vector  $s$ , we obtain a variant of `map` that is applicable to any integer array, no matter whether it is a scalar, a vector, a matrix, or anything else. It is noteworthy that this generalization does not require to change the definition of `map` any further.

```
map :  $\Pi r :: \text{nat}. \Pi s :: \text{natvec}(r).$ 
      (int  $\rightarrow$  int)  $\rightarrow$  [int | s]  $\rightarrow$  [int | s]
map 'r 's f a = gen x < shape a of [ | [0] |] with f a.[x]
```

To provide an example that makes use of non-scalar array cells, we define multiplication for arrays of complex numbers. We represent complex numbers as two-element vectors of doubles, stored in the cells of a double array. Thus, a complex array of shape  $s$  is represented by a double array of shape  $s ++ [2]$ . For each complex product, the program `cpxm1` selects the real and imaginary parts of the corresponding numbers from the argument arrays. The resulting complex number becomes a cell in the result array.

```

cpxmul : Πr::nat. Πs::natvec(r).
  [double|s++[2]] → [double|s++[2]] → [double|s++[2]]
cpxmul 'r 's a b =
  gen x < take {rank a - 1, shape a} of [|2| [1] |] with
  let ar = a.[x++ [0]] in let ai = a.[x++ [1]] in
  let br = b.[x++ [0]] in let bi = b.[x++ [1]] in
  [ar*br - ai*bi, ar*bi + ai*br| [2]]

```

The generalized selection `gsel` overcomes the restriction that the length of a selection vector must match the rank of the array selected into. Given an array  $a$  and selection vector  $x$  whose length is shorter than the rank of  $a$ , it selects an array slice with those elements whose position in  $a$  is prefixed with  $x$ . The shape of the result is thus `drop {length x, shape a}`. We use a singleton type for the selection vector to enforce that its value must range between  $\bar{0}$  and a prefix of the array shape.

```

gsel : Πr::nat. Πs::natvec(r).
  Πl::[nat in ..r+1]. Πv::{natvec(l) in ..take(l,s)}.
  [int|s] → numvec(v) → [int|drop(l,s)]
gsel 'r 's 'l 'v a x = gen y < drop {length x, shape a} of [| | [0] |]
  with a.[x+y]

```

Another interesting example is `iota`, a function that combines the power of singleton types with dependent pairs. Given a non-negative integer vector  $v$ , `iota` yields an array that contains all valid index vectors into an array of shape  $v$ . The  $\Sigma$ -type indicates precisely that the values of the vectors range between  $\bar{0}$  and  $v$ .

```

iota : Πr::nat. Πs::natvec(r).
  numvec(s) → [Σy::{natvec(r) in ..s}. numvec(y) | s]
iota 'r 's v = gen x < v of [| | [0] |]
  with { 'x, x : Σy::{natvec(r) in ..s}. numvec(y)}

```

The result of `iota` can for example be used with the multiple selection `mselect`. It takes an array  $a$  and another array  $i$  of (legal) selection vectors into  $a$ . `mselect` then performs a selection into  $a$  for every vector in  $i$  and yields the array of all results.

```

mselect : Πr::nat. Πs::natvec(r).
  Πt::nat. Πu::natvec(t).
  [int|s] → [Σy::{natvec(r) in ..s}. numvec(y) | u] →
  [int|u]
mselect 'r 's 't 'u a i = gen x < shape i of [| | [0] |]
  with let { 'j, y } = i.[x] in a.[y]

```

Using loops, we can define shape-generic variants of the well-known higher-order `fold` functions. While `foldl` traverses the array elements in lexicographic order, `foldr` starts with the greatest array index and progresses in descending order.

```

foldl : Πr::nat. Πs::natvec(r).
  (int → int → int) → int → [int|s] → int
foldl 'r 's f n a =
  loop x < shape a, acc = n with (f acc a.[x])

foldr : Πr::nat. Πs::natvec(r).
  (int → int → int) → int → [int|s] → int
foldr 'r 's f n a =
  let as = shape a in
  let b = as - (vec {length as, 1}) in
  loop x < shape a, acc = n with (f a.[b-x] acc)

```

#### 4.2. Case study: inner product

As a more elaborate example for the expressive power of shape-generic functional array programming, we now present a program for computing matrix products. We will then generalize this program with little effort such that it can also be used to compute matrix-vector products, vector-vector products and similar operations.

Matrix multiplication is a shape-generic function with complex constraints on the shapes of its arguments. Only if the number of columns of the first matrix equals the number of rows of the second matrix, the result matrix will have as many rows as the first argument and as many columns as the second.

```

matmul : Πp::natvec(1). Πq::natvec(1). Πr::natvec(1).
  [int|p+q] → [int|q+r] → [int|p+r]

```

We implement matrix multiplication by means of a `WITH`-loop that for each element of the result array fetches the corresponding row from the first argument and the column from the second argument. It then combines both vectors into a scalar by element-wise multiplication and subsequent reduction by summation.

$$\begin{array}{c}
\Gamma \vdash \text{idx} :: *_I \text{ (WFS-IDX)} \\
\frac{\Gamma \vdash i :: \{\text{idx in } 0.. \}}{\Gamma \vdash \text{idxvec}(i) :: *_I} \text{ (WFS-VEC)} \\
\frac{\Gamma \vdash I :: *_I \quad \Gamma \vdash ir :: I_r \quad \Gamma, x :: I \vdash x :: I_r}{\Gamma \vdash \{I \text{ in } ir\} :: *_I} \text{ (WFS-SUBSET)}
\end{array}$$

**Fig. 9.** Well-formedness of sorts.

```

matmul 'p'q'r a b =
  let pp = take {1, shape a} in
  let rr = drop {1, shape b} in
  gen x < pp ++ rr of [| [0] |] with
    let arow = gsel '2'(p++q)'1'(take(1,x)) a (take {1,x}) in
    let bcol = fsel '2'(q+r)'1'(drop(1,x)) b (drop {1,x}) in
    sum '1'q (mul '1'q arow bcol)

```

In addition to the generalized selection `gsel` for selecting rows, the program uses a similar function called `fsel` for selecting columns. The function `sum` is defined in terms of `foldl`. In the definition of `mul` we assume we have an infix operator `*` for computing the integer product.

```

fsel : Πr::nat. Πs::natvec(r).
      Πl1::{nat in ..r+1}. Πv::{natvec(1) in ..drop(r-1,s)}.
      [int|s] → numvec(v) → [int|take(r-1,s)]
fsel 'r's'l'v a x =
  gen y < take {(rank a) - (length x), shape a} of [| [0] |]
  with a.[y++x]

sum : Πr::nat. Πs::natvec(r). [int|s] → int
sum 'r's a = foldl 'r's (λx:int. λy:int. (x+y)) 0 a

mul : Πr::nat. Πs::natvec(r). [int|s] → [int|s] → [int|s]
mul 'r's a b = gen x < shape a of [| [0] |] with a.[x]*b.[x]

```

An interesting generalization of the matrix multiplication scheme is the inner product `ip`. Instead of restricting its arguments to (suitable) matrices, `ip` allows the arguments to have arbitrary shapes and an arbitrary number of axes as long as the last axis of the first argument is as long as the first axis of the second argument. The inner product then combines all the vectors along the last axis (rows) of the first array with all vectors along the first axis (columns) of the second array similar to the matrix multiplication. The algorithm for the inner product can be obtained from the matrix multiplication with minimal effort by simply adding index parameters for the array ranks.

```

ip : Πd::nat. Πe::nat.
     Πp::natvec(d). Πq::natvec(1). Πr::natvec(e).
     [int|p++q] → [int|q++r] → [int|p++r]
ip 'd'e'p'q'r a b =
  let dd = (rank a) - 1 in
  let pp = take {dd, shape a} in
  let rr = drop {1, shape b} in
  gen x < pp ++ rr of [| [0] |] with
    let arow = gsel '(d+1)'(p++q)'d'(take(d,x)) a (take {dd,x}) in
    let bcol = fsel '(e+1)'(q++r)'e'(drop(d,x)) b (drop {dd,x}) in
    sum '1'q (mul '1'q arow bcol)

```

Having defined the algorithm for the shape-generic inner product, we may derive rank-specific algorithms for matrix multiplication or matrix–vector products by partial application:

```

matmul = ip '1'1
matvecmul = ip '1'0
sprod = ip '0'0

```

## 5. Type checking

The evaluation rules will only evaluate array terms under certain constraints between ranks, shape vectors, and even array elements. To rule out programs that will not evaluate to a value, we now present a type system for static verification

$$\begin{array}{c}
\frac{\Gamma \vdash i :: \{I \text{ in } ir\}}{\Gamma \vdash i :: I} \text{ (S-SUPERSET)} \\
\frac{\Gamma \vdash i :: \text{idx} \quad \Gamma \vdash i :: I \quad \Gamma \models i \text{ in } ir}{\Gamma \vdash i :: \{I \text{ in } ir\}} \text{ (S-SSUBSET)} \\
\frac{\Gamma \vdash i :: \text{idxvec}(i_j) \quad \Gamma \vdash i :: I \quad \Gamma \models i \text{ in } ir}{\Gamma \vdash i :: \{I \text{ in } ir\}} \text{ (S-VSUBSET)} \\
\frac{\Gamma \vdash i :: \text{idxvec}(i_1) \quad \Gamma \vdash i_2 :: \{\text{idx in } i_1\}}{\Gamma \vdash i :: \text{idxvec}(i_2)} \text{ (S-VLEN)} \\
\frac{x :: I \in \Gamma}{\Gamma \vdash x :: I} \text{ (S-CTX)} \\
\Gamma \vdash c :: \text{idx} \text{ (S-IDX)} \\
\frac{\forall j. \Gamma \vdash i_j :: \text{idx}}{\Gamma \vdash [i_1, \dots, i_n] :: \text{idxvec}(n)} \text{ (S-VECT)} \\
\frac{\Gamma \vdash i_1 :: \{\text{idx in } 0..n\} \quad \Gamma \vdash i_2 :: \text{idx}}{\Gamma \vdash \text{vec}(i_1, i_2) :: \text{idxvec}(i_1)} \text{ (S-VEC)} \\
\frac{\Gamma \vdash i_1 :: \text{idxvec}(m) \quad \Gamma \vdash i_2 :: \text{idxvec}(n)}{\Gamma \vdash i_1 \# i_2 :: \text{idxvec}(m+n)} \text{ (S-CAT)} \\
\frac{\Gamma \vdash i_1 :: \{\text{idx in } 0..n+1\} \quad \Gamma \vdash i_2 :: \text{idxvec}(n)}{\Gamma \vdash \text{take}(i_1, i_2) :: \text{idxvec}(i_1)} \text{ (S-TAKE)} \\
\frac{\Gamma \vdash i_1 :: \{\text{idx in } 0..n+1\} \quad \Gamma \vdash i_2 :: \text{idxvec}(n)}{\Gamma \vdash \text{drop}(i_1, i_2) :: \text{idxvec}(n-i_1)} \text{ (S-DROP)} \\
\frac{\Gamma \vdash i_1 :: \text{idx} \quad \Gamma \vdash i_2 :: \text{idx}}{\Gamma \vdash f_2(i_1, i_2) :: \text{idx}} \text{ (S-SBIN)} \\
\frac{\Gamma \vdash i_1 :: \text{idxvec}(i) \quad \Gamma \vdash i_2 :: \text{idxvec}(i)}{\Gamma \vdash f_2(i_1, i_2) :: \text{idxvec}(i)} \text{ (S-VBIN)} \\
\frac{\Gamma \vdash i :: \text{idx}}{\Gamma \vdash i.. :: \text{idx}} \text{ (RS-SFROM)} \quad \frac{\Gamma \vdash i :: \text{idx}}{\Gamma \vdash ..i :: \text{idx}} \text{ (RS-STO)} \\
\frac{\Gamma \vdash i_1 :: \text{idx} \quad \Gamma \vdash i_2 :: \text{idx}}{\Gamma \vdash i_1..i_2 :: \text{idx}} \text{ (RS-SFROMTO)} \\
\frac{\Gamma \vdash i :: \text{idxvec}(i_j)}{\Gamma \vdash i.. :: \text{idxvec}(i_j)} \text{ (RS-VFROM)} \quad \frac{\Gamma \vdash i :: \text{idxvec}(i_j)}{\Gamma \vdash ..i :: \text{idxvec}(i_j)} \text{ (RS-VTO)} \\
\frac{\Gamma \vdash i_1 :: \text{idxvec}(i_j) \quad \Gamma \vdash i_2 :: \text{idxvec}(i_j)}{\Gamma \vdash i_1..i_2 :: \text{idxvec}(i_j)} \text{ (RS-VFROMTO)}
\end{array}$$

Fig. 10. The sorting relation.

of array programs. Besides the terms, array programs also contain index terms as well as sort and type declarations. Thus, in addition to type checking the terms, we must sort check the index terms and verify the declarations' well-formedness.

We specify the typing rules in a declarative style. This style makes the rules short and clear. However, it also allows rules to be applied in non-deterministic order and may result in potentially infinite typing derivations. We briefly sketch out how the rules may be adapted for obtaining a type checking algorithm at the end of the chapter.

### 5.1. Typing context

All relations necessary for verifying array programs employ a common typing context  $\Gamma$ . It includes type declarations  $x : T$ , sort declarations  $x :: I$ , and additional constraints for confining index terms to specific index ranges, e.g.  $x + 1 \text{ in } 0..10$ . We assume that all variable names are pairwise distinct and that all types, sorts, and index terms used in the context are well-formed. In particular, all index variables used in a specific context element must have been declared earlier.

$$\Gamma ::= \cdot \mid \Gamma, x : T \mid \Gamma, x :: I \mid \Gamma, i \text{ in } ir$$

### 5.2. Semantic judgments

During type checking, it is often necessary to verify that the value denoted by an index term only ranges within specific bounds. We employ the two judgments  $\Gamma \models i \text{ in } ir$  and  $\Gamma \models i \text{ in } ir$  to prove such propositions for scalar indices and for index vectors, respectively: Both judgments are decided outside of the type system with decision procedures working on

$$\begin{array}{c}
\Gamma \vdash \text{int} : *_Q \text{ (QWF-INT)} \\
\frac{\Gamma \vdash T_1 : * \quad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \rightarrow T_2 : *_Q} \text{ (QWF-FUN)} \\
\frac{\Gamma \vdash I :: *_I \quad \Gamma, x :: I \vdash T : *}{\Gamma \vdash \Pi x :: I. T : *_Q} \text{ (QWF-PI)} \\
\frac{\forall j. \Gamma \vdash T_j : *}{\Gamma \vdash \{T^n\} : *_Q} \text{ (QWF-TUP)} \\
\frac{\Gamma \vdash I :: *_I \quad \Gamma, x :: I \vdash T : *}{\Gamma \vdash \Sigma x :: I. T : *_Q} \text{ (QWF-SIGMA)} \\
\frac{\Gamma \vdash Q : *_Q \quad \Gamma \vdash i :: \{\text{idxvec}(n) \text{ in } \text{vec}(n, 0) \dots\}}{\Gamma \vdash [Q|i] : *} \text{ (WF-ARRAY)} \\
\frac{\Gamma \vdash i :: \text{idx}}{\Gamma \vdash \text{num}(i) : *} \text{ (WF-NUM)} \\
\frac{\Gamma \vdash i :: \text{idxvec}(n)}{\Gamma \vdash \text{numvec}(i) : *} \text{ (WF-NUMVEC)}
\end{array}$$

**Fig. 11.** Well-formedness of types and quark types.

the interpretation of the sorts  $\text{idx}$  and  $\text{idxvec}(i)$  as integers and vectors of integers. We will describe these procedures in Section 6. Using the index judgment for vectors, we may, for example, verify that a vector of positive numbers is also non-negative:

$$r :: \{\text{idx in } 0 \dots\}, s :: \{\text{idxvec}(r) \text{ in } \text{vec}(r, 1) \dots\} \models s \text{ in } \text{vec}(r, 0) \dots$$

### 5.3. Well-formedness of sorts

Fig. 9 shows the relation  $\Gamma \vdash I :: *_I$  for checking well-formedness of index sorts. Using the sorting relation  $\Gamma \vdash i :: I$ , the rule WFS-VEC ensures that, for every vector sort  $\text{idxvec}(i)$ ,  $i$  is a non-negative integer. The rule WFS-SUBSORT accepts only those subset sorts  $\{I \text{ in } ir\}$  whose bounds in  $ir$  have a sort compatible with the base sort  $I$ , i.e. they have a common root sort  $I_r$ .

### 5.4. Sort checking

Every index term has an infinite number of sorts. For example, the index term  $1 + 1$  may, as any scalar index, have the sort  $\text{idx}$ . However, it is also a natural number  $\{\text{idx in } 0 \dots\}$ , a number between 0 and 10  $\{\text{idx in } 0 \dots 10\}$ , and an integer equal to 2  $\{\text{idx in } 2\}$ .

The rules at the top of the sorting relation shown in Fig. 10 formalize this intuition. The rule S-SUPERSET states that every index of sort  $\{I \text{ in } ir\}$  is also of sort  $I$ . Conversely, if we can prove that an index term  $i$  of sort  $I$  is constrained by a range  $ir$  then it is also of sort  $\{I \text{ in } ir\}$ . Depending on whether  $i$  is a scalar or a vector, the rules S-SSUBSET and S-VSUBSET will prove the constraint using the scalar or the vector judgment, respectively. It is noteworthy that there are no other rules employing the constraint provers. The rule S-VLEN uses this machinery to identify vector sorts of equal lengths, e.g. a vector of sort  $\text{idxvec}(1 + 2)$  also has sort  $\text{idxvec}(3)$ .

The rules for checking index terms determine for each term a general sort according to the term's meaning as described in Section 3 while ensuring only the necessary preconditions. The remaining rules in the figure define an auxiliary sorting relation  $\Gamma \vdash ir :: I$  for checking the well-formedness of index ranges.

### 5.5. Well-formedness of types

The well-formedness relations for quark types  $\Gamma \vdash Q : *_Q$  and types  $\Gamma \vdash T : *$  are shown in Fig. 11. The relations follow the mutually recursive structure of the types. A quark type is well-formed if the types and sorts it refers to are well-formed. Similarly, an array type  $[Q|i]$  is well-formed if  $Q$  is a well-formed quark type and the index expression  $i$  denotes a non-negative vector. The type of singleton scalars  $\text{num}(i)$  requires a scalar index term  $i$ , whereas singleton vector types  $\text{numvec}(i)$  need an index vector. Note that  $\perp_Q$  is not a well-formed quark type: it may arise during type-checking but the programmer is not allowed to use it explicitly in a program.

$$\begin{array}{c}
\Gamma \vdash Q <:_{\mathcal{Q}} Q \text{ (QSUB-REFL)} \\
\frac{\Gamma \vdash Q_1 <:_{\mathcal{Q}} Q_2 \quad \Gamma \vdash Q_2 <:_{\mathcal{Q}} Q_3}{\Gamma \vdash Q_1 <:_{\mathcal{Q}} Q_3} \text{ (QSUB-TRANS)} \\
\Gamma \vdash \perp_{\mathcal{Q}} <:_{\mathcal{Q}} Q \text{ (QSUB-BOT)} \\
\frac{\Gamma \vdash S_1 <: T_1 \quad \Gamma \vdash T_2 <: S_2}{\Gamma \vdash T_1 \rightarrow T_2 <:_{\mathcal{Q}} S_1 \rightarrow S_2} \text{ (QSUB-FUN)} \\
\frac{\Gamma, x :: I_2 \vdash x :: I_1 \quad \Gamma, x_2 :: I_2 \vdash T_1[x_1 \mapsto_i x_2] <: T_2}{\Gamma \vdash \Pi x_1 :: I_1. T_1 <:_{\mathcal{Q}} \Pi x_2 :: I_2. T_2} \text{ (QSUB-PI)} \\
\frac{\forall j. \Gamma \vdash T_j <: S_j}{\Gamma \vdash \{T^n\} <:_{\mathcal{Q}} \{S^n\}} \text{ (QSUB-TUP)} \\
\frac{\Gamma, x :: I_1 \vdash x :: I_2 \quad \Gamma, x_1 :: I_1 \vdash T_1 <: T_2[x_2 \mapsto_i x_1]}{\Gamma \vdash \Sigma x_1 :: I_1. T_1 <:_{\mathcal{Q}} \Sigma x_2 :: I_2. T_2} \text{ (QSUB-SIGMA)} \\
\Gamma \vdash T <: T \text{ (SUB-REFL)} \\
\frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \text{ (SUB-TRANS)} \\
\frac{\Gamma \vdash Q_1 <:_{\mathcal{Q}} Q_2}{\Gamma \vdash [Q_1 | i] <: [Q_2 | i]} \text{ (SUB-ARRQ)} \\
\frac{\Gamma \vdash i_1 :: \text{idxvec}(i) \quad \Gamma \vdash i_2 :: \{\text{idxvec}(i) \text{ in } i_1\}}{\Gamma \vdash [Q | i_1] <: [Q | i_2]} \text{ (SUB-ARRSHP)} \\
\frac{\Gamma \vdash i_1 :: I \quad \Gamma \vdash i_2 :: \{I \text{ in } i_1\}}{\Gamma \vdash S(i_1) <: S(i_2)} \text{ (SUB-SINGLE)} \\
\Gamma \vdash \text{num}(i) <: [\text{int} | []] \text{ (SUB-NUM)} \\
\frac{\Gamma \vdash i :: \text{idxvec}(i)}{\Gamma \vdash \text{numvec}(i) <: [\text{int} | [I]]} \text{ (SUB-NUMVEC)}
\end{array}$$

Fig. 12. Subtyping on types and quark types.

### 5.6. Subtyping

The subtype relations on types  $\Gamma \vdash T <: T$  and quark types  $\Gamma \vdash Q <:_{\mathcal{Q}} Q$ , shown in Fig. 12, follow the same mutually recursive pattern. Both relations are reflexive and transitive. The bottom quark type  $\perp_{\mathcal{Q}}$  is a subtype of every quark type. As in other type systems, subtyping on function quark types is contravariant in the argument type and covariant in the result type (QSUB-FUN). More generally, according to QSUB-PI, a dependent function quark type  $\Pi x_1 :: I_1. T_1$  is a subtype of another dependent function type  $\Pi x_2 :: I_2. T_2$  if two conditions are met: Firstly,  $I_2$  must denote a subset of  $I_1$ . This is verified by declaring a fresh variable  $x$  of sort  $I_2$  and deriving that  $x$  then also has sort  $I_1$ . Secondly, when applied to an argument of sort  $I_2$ , the result of the first function must have a type which is a subtype of the second function's result type. The subtype relation for both the tuple quark type  $\{T^n\}$  and the dependent pair quark type  $\Sigma x :: I. T$  is covariant in all positions.

The rules SUB-NUM and SUB-NUMVEC formalize that every singleton scalar is also a scalar integer array and that a singleton vector is also an integer vector. Subtyping on array types is covariant: by SUB-ARRQ, an array type  $[Q_1 | i]$  is a subtype of another array type  $[Q_2 | i]$  when  $Q_1$  is a subtype of  $Q_2$ . This intuitive subtyping rule is known to cause problems in the presence of mutable arrays [12]: An array of type  $[Q_1 | i]$  may be known in a different context as a  $[Q_2 | i]$ , with  $\Gamma \vdash Q_1 <:_{\mathcal{Q}} Q_2$ . Now, updating an element in the latter context with a quark of type  $Q_2$  will break the typing in the former context. It is a clear advantage of immutable arrays that they are not affected by this subtle issue. The array types  $[Q | i_1]$  and  $[Q | i_2]$  are equivalent by rule SUB-ARRSHP if  $i_1$  and  $i_2$  denote the same shape. SUB-SINGLE defines a similar equality for singleton types.

### 5.7. Type checking

Now that we have treated all the prerequisites, we can define the typing relation  $\Gamma \vdash t : T$  and the quark typing relation  $\Gamma \vdash q :_{\mathcal{Q}} Q$ . The most basic typing rules for functional array programs are summarized in Fig. 13. The subsumption rules QT-SUB and T-SUB state that quarks and terms have multiple types through subtyping.

$$\begin{array}{c}
\frac{\Gamma \vdash q :_Q Q_1 \quad \Gamma \vdash Q_1 <:_Q Q_2}{\Gamma \vdash q :_Q Q_2} \text{ (QT-SUB)} \\
\Gamma \vdash c :_Q \text{int} \text{ (QT-INT)} \\
\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1. t :_Q T_1 \rightarrow T_2} \text{ (QT-Abs)} \\
\frac{\Gamma, x :: I \vdash t : T}{\Gamma \vdash \lambda' x :: I. t :_Q \Pi x :: I. T} \text{ (QT-PI)} \\
\frac{\forall j. \Gamma \vdash v_j : T_j}{\Gamma \vdash \{v^n\} :_Q \{T^n\}} \text{ (QT-TUP)} \\
\frac{\Gamma \vdash \Sigma x :: I. T : *_Q \quad \Gamma \vdash i :: I \quad \Gamma \vdash t : T[x \mapsto i]}{\Gamma \vdash \{i, t : \Sigma x :: I. T\} :_Q \Sigma x :: I. T} \text{ (QT-SIGMA)} \\
\frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash t : T_2} \text{ (T-SUB)} \\
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-Ctx)} \\
\frac{n > 0 \quad \forall j. \Gamma \vdash q_j :_Q Q}{\Gamma \vdash [q^n | [s^d] ] : [Q | [s^d] ]} \text{ (T-VAL)} \\
\Gamma \vdash [ | [s^d] ] : [\perp_Q | [s^d] ] \text{ (T-VALE)} \\
\Gamma \vdash [c | [] ] : \text{num}(c) \text{ (T-NUM)} \\
\Gamma \vdash [c^n | [n] ] : \text{numvec}([c^n]) \text{ (T-NUMVEC)} \\
\frac{\Gamma \vdash t_1 : [T_1 \rightarrow T_2 | [] ] \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2} \text{ (T-APP)} \\
\frac{\Gamma \vdash t : [\Pi x :: I. T | [] ] \quad \Gamma \vdash i :: I}{\Gamma \vdash t' i : T[x \mapsto i]} \text{ (T-IAPP)} \\
\frac{\forall j. \Gamma \vdash t_j : T_j}{\Gamma \vdash \{t^n\} : [\{T^n\} | [] ]} \text{ (T-TUP)} \\
\frac{\Gamma \vdash \Sigma x :: I. T : *_Q \quad \Gamma \vdash i :: I \quad \Gamma \vdash t : T[x \mapsto i]}{\Gamma \vdash \{i, t : \Sigma x :: I. T\} : [\Sigma x :: I. T | [] ]} \text{ (T-ITUP)} \\
\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \text{ (T-LET)} \\
\frac{\Gamma \vdash t_1 : [\{T^n\} | [] ] \quad \Gamma, x_1 : T_1, \dots, x_n : T_n \vdash t_2 : T_{n+1}}{\Gamma \vdash \text{let } \{x^n\} = t_1 \text{ in } t_2 : T_{n+1}} \text{ (T-UNPACK)} \\
\frac{\Gamma \vdash t_1 : [\Sigma x :: I. T | [] ] \quad \Gamma, x_i :: I, x : T[x \mapsto x_i] \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{x_i, x\} = t_1 \text{ in } t_2 : T_2} \text{ (T-IUNPACK)}
\end{array}$$

Fig. 13. Basic typing rules.

According to rule T-VAL, type checking of non-empty array values  $[q^n | [s^d] ]$  requires to verify that each quark  $q_i$  has the same quark type  $Q$ . For arrays of abstractions,  $Q$  has the form  $T_1 \rightarrow T_2$ . Using the declared domain type  $T_1$ , the rule QT-Abs, checks an abstraction quark  $\lambda x : T_1. t$  by inserting  $x : T_1$  into the environment and determining its result type  $T_2$ . The rule for dependent functions works analogously. A dependent pair  $\{i, t : \Sigma x :: I. T\}$  has the quark type  $\Sigma x :: I. T$  if the index term  $i$  has sort  $I$  and if the term  $t$  has the type obtained by substituting all references to the identifier  $x$  in  $T$  with the index term  $i$ .

For an empty array value without quarks, no precise quark type can be determined. For this reason, rule T-VALE assigns it the bottom quark type  $\perp_Q$ , which is a quark subtype of any quark type. In addition to their array types, constant integer scalars and vectors also have more specific constant singleton types.

The rules T-APP and T-IAPP ensure that only scalar arrays of (dependent) functions can be applied to suitable arguments. The result of applying a dependent function of type  $\Pi x :: I. T$  to an index  $i$  has type  $T$  in which all index identifiers  $x$  have



$$\begin{array}{c}
\frac{\Gamma \vdash t : [Q | i] \quad \Gamma \vdash i :: \text{idxvec}(i)}{\Gamma \vdash \text{rank } t : \text{num}(i)} \text{ (T-RANK)} \\
\frac{\Gamma \vdash t : [Q | i]}{\Gamma \vdash \text{shape } t : \text{numvec}(i)} \text{ (T-SHAPE)} \\
\frac{\Gamma \vdash t : \text{numvec}(i) \quad \Gamma \vdash i :: \text{idxvec}(i)}{\Gamma \vdash \text{length } t : \text{num}(i)} \text{ (T-LENGTH)} \\
\frac{\Gamma \vdash t : [\{\text{num}(i_1), \text{num}(i_2)\} | []]}{\Gamma \vdash f_2 t : \text{num}(f_2(i_1, i_2))} \text{ (T-BINS)} \\
\frac{\Gamma \vdash t : [\{\text{numvec}(i_1), \text{numvec}(i_2)\} | []] \quad \Gamma \vdash i_1 :: \text{idxvec}(i) \quad \Gamma \vdash i_2 :: \text{idxvec}(i)}{\Gamma \vdash f_2 t : \text{numvec}(f_2(i_1, i_2))} \text{ (T-BINV)} \\
\frac{\Gamma \vdash t : [\{\{\text{int} | i\}, [\text{int} | i]\} | []]}{\Gamma \vdash f_2 t : [\text{int} | i]} \text{ (T-BIN)} \\
\frac{\Gamma \vdash t : [\{\{[Q | i_s], \text{numvec}(i)\} | []] \quad \Gamma \vdash i_s :: \text{idxvec}(i) \quad \Gamma \vdash i :: \{\text{idxvec}(i) \text{ in } \text{vec}(i, 0) \dots i_s\}}{\Gamma \vdash \text{sel } t : [Q | []]} \text{ (T-SEL)} \\
\frac{\Gamma \vdash t : [\{\text{num}(i), \text{num}(i)\} | []] \quad \Gamma \vdash i :: \{\text{idx in } 0 \dots\}}{\Gamma \vdash \text{vec } t : \text{numvec}(\text{vec}(i, i))} \text{ (T-VEC)} \\
\frac{\Gamma \vdash t : [\{\text{numvec}(i_1), \text{numvec}(i_2)\} | []]}{\Gamma \vdash ++ t : \text{numvec}(i_1 ++ i_2)} \text{ (T-CAT)} \\
\frac{\Gamma \vdash t : [\{\text{num}(i), \text{numvec}(i_v)\} | []] \quad \Gamma \vdash i_v :: \text{idxvec}(i) \quad \Gamma \vdash i :: \{\text{idx in } 0 \dots i + 1\}}{\Gamma \vdash \text{take } t : \text{numvec}(\text{take}(i, i_v))} \text{ (T-TAKE)} \\
\frac{\Gamma \vdash t : [\{\text{num}(i), \text{numvec}(i_v)\} | []] \quad \Gamma \vdash i_v :: \text{idxvec}(i) \quad \Gamma \vdash i :: \{\text{idx in } 0 \dots i + 1\}}{\Gamma \vdash \text{drop } t : \text{numvec}(\text{drop}(i, i_v))} \text{ (T-DROP)} \\
\frac{\forall j. \Gamma \vdash t_j : [Q | i]}{\Gamma \vdash [t^j | [c^j]] : [Q | [c^j] ++ i]} \text{ (T-ARR)} \\
\frac{\forall j. \Gamma \vdash t_j : \text{num}(i_j)}{\Gamma \vdash [t^j | [n]] : \text{numvec}([n^j])} \text{ (T-ARRNUMVEC)} \\
\frac{\Gamma \vdash t_1 : \text{numvec}(i_1) \quad \Gamma \vdash i_1 :: \{\text{idxvec}(n) \text{ in } \text{vec}(n, 0) \dots\} \quad \Gamma \vdash t_2 : \text{numvec}(i_2) \quad \Gamma \vdash i_2 :: \{\text{idxvec}(m) \text{ in } \text{vec}(m, 0) \dots\} \quad \Gamma, x :: \{\text{idxvec}(n) \text{ in } \text{vec}(n, 0) \dots i_1\}, x : \text{numvec}(x) \vdash t_3 : [Q | i_2]}{\Gamma \vdash \text{gen } x < t_1 \text{ of } t_2 \text{ with } t_3 : [Q | i_1 ++ i_2]} \text{ (T-GEN)} \\
\frac{\Gamma \vdash t_1 : \text{numvec}(i) \quad \Gamma \vdash i :: \{\text{idxvec}(n) \text{ in } \text{vec}(n, 0) \dots\} \quad \Gamma \vdash t_2 : T \quad \Gamma, x_1 :: \{\text{idxvec}(n) \text{ in } \text{vec}(n, 0) \dots i\}, x_1 : \text{numvec}(x_1), x_2 : T \vdash t_3 : T}{\Gamma \vdash \text{loop } x_1 < t_1, x_2 = t_2 \text{ with } t_3 : T} \text{ (T-LOOP)}
\end{array}$$

**Fig. 14.** Typing rules for the array-specific language elements.

been replaced with  $i$ . Well-typed tuple and dependent pair constructors yield scalar arrays containing the respective quark. Vice versa, unpacking can only be performed for scalar tuples.

Typing of the array specific built-ins is shown in Fig. 14. The rank and shape primitives can be applied to arbitrary arrays and yield singleton types. `length` is only applicable to singleton vectors and yields a scalar singleton. Three rules are used to type applications of binary operations: They may be applied to integer arrays of equal shape (T-BIN), yielding another of the same element type and shape. More interestingly, when applied to (compatible) singletons (T-BINS, T-BINV), the result is also a singleton whose value is characterized by the application of the operation to the original singletons' indices. The vector operations `vec`, `take`, and `drop` always require appropriate singleton arguments and yield a singleton vector formed in the same way.

The typing rule T-SEL statically enforces all the necessary preconditions of the selection: the selection vector must be a singleton with appropriate length that ranges within the boundaries of the array selected into. A (valid) selection always yields a scalar array but never a singleton.

An array constructor with frame shape  $f$  is well-typed if all cells have the same quark type  $Q$  and the same shape  $i_c$ . The new array then has type  $[Q | f ++ i_c]$ . In the special case where all cells of a vector are singleton scalars, rule T-ARRNUMVEC

$$\begin{array}{c}
\frac{\Gamma \vdash t : S(i) \quad \Gamma | S(i) \vdash m : T_m}{\Gamma \vdash \text{case } t \text{ in } m : T_m} \text{ (T-CASE)} \\
\frac{\Gamma \vdash t : T}{\Gamma | S(i) \vdash \text{else} \Rightarrow t : T} \text{ (T-ELSE)} \\
\frac{\Gamma | S(i) \vdash r \# i : ir \quad \Gamma, i \text{ in } ir \vdash t : T \quad \Gamma | S(i) \vdash m : T}{\Gamma | S(i) \vdash r \Rightarrow t | m : T} \text{ (T-RANGE)} \\
\frac{\Gamma \vdash t : S(i_r) \quad \Gamma \vdash i_r :: I \quad \Gamma \vdash i :: I}{\Gamma | S(i) \vdash t ::_r i_r} \text{ (IR-EQ)} \\
\frac{\Gamma \vdash t : S(i_r) \quad \Gamma \vdash i_r :: I \quad \Gamma \vdash i :: I}{\Gamma | S(i) \vdash t \dots ::_r i_r \dots} \text{ (IR-FROM)} \\
\frac{\Gamma \vdash t : S(i_r) \quad \Gamma \vdash i_r :: I \quad \Gamma \vdash i :: I}{\Gamma | S(i) \vdash \dots t \dots ::_r \dots i_r} \text{ (IR-TO)} \\
\frac{\Gamma \vdash t_1 : S(i_1) \quad \Gamma \vdash t_2 : S(i_2) \quad \Gamma \vdash i_1 :: I \quad \Gamma \vdash i_2 :: I \quad \Gamma \vdash i :: I}{\Gamma | S(i) \vdash t_1 \dots t_2 ::_r i_1 \dots i_2} \text{ (IR-FROMTO)}
\end{array}$$

Fig. 15. Typing rules for conditional expressions.

gives the array the appropriate singleton vector type. Typing of a WITH-loop  $\text{gen } x < t_1 \text{ of } t_2 \text{ with } t_3$  verifies that the frame shape  $t_1$  and the cell shape  $t_2$  are non-negative vectors associated with the index vectors  $i_1$  and  $i_2$ , respectively. For checking the cell expression  $t_3$ , the identifier  $x$  is bound to both a vector sort ranging between zero and the frame shape and a singleton vector with exactly that value. If the cell expression then has type  $[Q | i_2]$ , where  $i_2$  is also the value of the cell shape  $t_2$ , then the WITH-loop has type  $[Q | i_1 \# i_2]$ .

Similarly, typing of a loop  $\text{loop } x_1 < t_1, x_2 = t_2 \text{ with } t_3$  also requires that the loop boundary  $t_1$  is a non-negative singleton vector. In addition to binding  $x_1$  to an appropriate sort and a singleton vector, the accumulator  $x_2$  is bound to the type of the initial value  $t_2$  during type checking of the loop expression  $t_3$ . If the loop expression preserves the accumulator's type, that type is also given to the entire loop.

Conditional expressions of the form  $\text{case } t \text{ in } m$  are typed according to the rules in Fig. 15. The type of the branching condition  $t$  is determined first and must be a singleton type. Its type is needed to verify that all ranges are compatible with the branching condition, i.e. that all ranges are integer singletons of the same shape as  $t$ . Therefore, the auxiliary typing relation  $\Gamma | S(i) \vdash m : T$  also takes the branching expression's type  $S(i)$ . For branches of the form  $r \Rightarrow t | m$ , the rule T-RANGE uses the range index relation  $\Gamma | S(i) \vdash r ::_r ir$  to check that the boundaries in  $r$  are indeed appropriate singletons denoting an index range  $ir$ . As a branch is only evaluated if the value of the branching condition lies within its range  $r$ , it is type checked with the additional property  $i \text{ in } ir$ . The branch must then have the same type as the other branches. The terminal branch  $\text{else} \Rightarrow t_e$  has the same type as  $t_e$ .

### 5.8. Properties of the type system

Having introduced all the rules, we can now prove that the type system indeed provides type-safety. For this, we have to show that each (closed) well-typed term is either a value or can make an evaluation step. Moreover, evaluation should preserve the well-typedness such that the term can be evaluated further. In our context, where we did not provide facilities for general recursion, this means that any well-typed array program will terminate yielding an array value.

**Theorem 5.1** (Progress). *For all closed and well-typed array terms  $t$ , either  $t$  is a value or  $\exists t'. t \longrightarrow t'$ .*

**Proof.** By induction on typing derivations (see proof in appendix of extended technical report [13]).  $\square$

**Theorem 5.2** (Preservation). *If  $\Gamma \vdash t : T$  and  $t \longrightarrow t'$ , then  $\Gamma \vdash t' : T$ .*

**Proof.** By induction on typing derivations (see proof in appendix of extended technical report [13]).  $\square$

We have specified the typing rules in a declarative style, which is concise, but does not allow for immediate implementation in a type checking algorithm. In particular, since neither index terms have a unique sort nor terms have a unique type, the sort and type conversion rules are applicable in non-deterministic order. In order to derive a decidable type checking algorithm,

the non-determinism must be tamed. Since defining an algorithmic set of typing rules is beyond the scope of this paper, we briefly sketch out the necessary modifications.

First, while most sort checking rules (Fig. 10) are syntax directed, the sort conversion rules apply in non-deterministic order. The sort conversion rules must be eliminated, their functionality transported into the all rules (not just those of the sorting relation) that require it. Second, subtyping (Fig. 12) introduces potential non-termination as the rules for transitivity and type equivalence apply arbitrarily. Via subsumption, these infinite derivations may arise anywhere in the typing derivation (Figs. 13–15). Thus, the subtyping rules must be replaced by an algorithm that checks whether a type is a subtype of another type. Instead of relying on subsumption, the typing scheme must apply this algorithm explicitly when necessary. Furthermore, without subsumption, bounded type joins and meets must be computed whenever a term's type depends on the types of more than one of its sub terms. Finally, more than one rule may apply for array values and array constructors. In these cases, preference must be given to the more special `num` and `numvec` types.

## 6. Resolving constraints

Type checking of array programs relies on the *semantic judgments*  $\Gamma \models i \text{ in } ir$  and  $\Gamma \bar{\models} i \text{ in } ir$ . They provide proof that under a given set of assumptions  $\Gamma$  the value denoted by an index term  $i$  is constrained to an interval  $ir$ . Both judgments are decided using procedures that operate on the interpretation of the index sorts `idx` and `idxvec(i)` as integers and vectors of integers.

We partition the context  $\Gamma$  into the set  $\mathcal{S}(\Gamma)$  which contains scalar sort declarations and properties and the set  $\mathcal{V}(\Gamma)$  consisting of vector sort declarations and constraints on vectors. Both sets do not contain sort declarations of subset sorts. These are transformed into a declaration of the root sort and a subsequent sequence of constraints, e.g.  $x :: \{\text{idx in } 0.. \} \rightsquigarrow x :: \text{idx}, x \text{ in } 0..$ . The type declarations in  $\Gamma$  are dispensable for constraint resolution. As shown in the example below, the scalar index terms in  $\mathcal{V}(\Gamma)$  may refer to variables from  $\mathcal{S}(\Gamma)$ . However, there is no converse dependency since no scalar term has a vector sub term.

$$\begin{aligned} \Gamma &= d :: \{\text{idx in } 0.. \}, s :: \{\text{idxvec}(d) \text{ in } \text{vec}(d, 1)..\}, x : [\text{int } |s] \\ \mathcal{S}(\Gamma) &= d :: \text{idx}, d \text{ in } 0.. \\ \mathcal{V}(\Gamma) &= s :: \text{idxvec}(d), s \text{ in } \text{vec}(d, 1).. \end{aligned}$$

Scalar judgments  $\Gamma \models i \text{ in } ir$  are checked using the assumptions in the set  $\mathcal{S}(\Gamma)$  only. The judgment is stated as a satisfiability problem with linear integer arithmetic by interpreting the index properties as linear inequalities. Current SMT solvers with support for linear arithmetic [14,15] can then refute the negated property, thereby validating the judgment.

$$\begin{aligned} d :: \text{idx}, d \text{ in } 0.., e :: \text{idx}, e \text{ in } d.. &\models e \text{ in } 0.. \\ \Leftrightarrow d \geq 0 \wedge e \geq d \wedge \neg e \geq 0 &\text{ **unsat**} \end{aligned}$$

The decision procedure for vector judgments  $\Gamma \bar{\models} i \text{ in } ir$  takes both sets  $\mathcal{S}(\Gamma)$  and  $\mathcal{V}(\Gamma)$  into account. Similar to the approach for scalars, we rewrite the problem such that it becomes verifiable with existing means. A straightforward approach would be to split up all vectors into scalar elements and to solve the resulting scalar formula. However, as the length of vectors typically depends on a variable bound in  $\mathcal{S}(\Gamma)$ , no finite number of elements will suffice. Thus, instead of rewriting the problem as a scalar formula, we state it as a formula in the array property fragment identified in [16] for which satisfiability is decidable.

An array property is a formula of the form  $\forall i. \varphi_I(i) \Rightarrow \varphi_V(i)$  where the *index guard*  $\varphi_I$  in our case always takes the form  $0 \leq i \wedge i \leq l - 1$  for some linear term denoting the vector length  $l$ . For readability, we write  $0 \leq i < l$ . In the *value constraint*, the quantified variable  $i$  may only be used in read expressions of the form  $a[i]$ .

The latter restriction rules out to express dependencies between a vector element at position  $i$  and another element at position  $j \neq i$ . For this reason, we cannot straightforwardly rewrite constraints between index vectors that contain the structural operations `take`, `drop`, or `++` as array properties. Scheme  $\mathcal{T}$  transforms well-behaved index vector terms into value constraint terms; Scheme  $\mathcal{P}$  transforms entire vector constraints into array properties, where  $|i|$  denotes the length of a vector term and each  $j$  is a fresh variable.

$$\begin{aligned} \mathcal{T} \llbracket x \rrbracket [i] &= x[i] \\ \mathcal{T} \llbracket s^t \rrbracket [i] &= s \\ \mathcal{T} \llbracket f_2(v_1, v_2) \rrbracket [i] &= f_2(\mathcal{T} \llbracket v_1 \rrbracket [i], \mathcal{T} \llbracket v_2 \rrbracket [i]) \\ \mathcal{P} \llbracket i_1 \text{ in } i_2 \rrbracket &= (\forall j. 0 \leq j < |i_1| \Rightarrow \mathcal{T} \llbracket i_1 \rrbracket [j] = \mathcal{T} \llbracket i_2 \rrbracket [j]) \\ \mathcal{P} \llbracket i_1 \text{ in } i_2.. \rrbracket &= (\forall j. 0 \leq j < |i_1| \Rightarrow \mathcal{T} \llbracket i_2 \rrbracket [j] \leq \mathcal{T} \llbracket i_1 \rrbracket [j]) \\ \mathcal{P} \llbracket i_1 \text{ in } ..i_2 \rrbracket &= (\forall j. 0 \leq j < |i_1| \Rightarrow \mathcal{T} \llbracket i_1 \rrbracket [j] < \mathcal{T} \llbracket i_2 \rrbracket [j]) \\ \mathcal{P} \llbracket i_1 \text{ in } i_2..i_3 \rrbracket &= (\forall j. 0 \leq j < |i_1| \Rightarrow \mathcal{T} \llbracket i_2 \rrbracket [j] \leq \mathcal{T} \llbracket i_1 \rrbracket [j] \wedge \mathcal{T} \llbracket i_1 \rrbracket [j] < \mathcal{T} \llbracket i_3 \rrbracket [j]) \end{aligned}$$

The following example shows a judgment for verifying that a vector of arbitrary length with strictly positive elements is also a non-negative vector and the corresponding satisfiability problem encoded in the array property fragment. As described

in [16], the quantifiers can be correctly eliminated from this formula by first converting into negated normal form and subsequently instantiating the universal quantifiers at appropriate index positions.

$$\begin{aligned}
& d :: \text{idx}, d \text{ in } 0.., s :: \text{idxvec}(d), s \text{ in } \text{vec}(d, 1) \dots \bar{\models} s \text{ in } \text{vec}(d, 0) \dots \\
& \Leftrightarrow d \geq 0 \wedge (\forall i. 0 \leq i < d \Rightarrow s[i] \geq 1) \wedge \neg(\forall i. 0 \leq i < d \Rightarrow s[i] \geq 0) \quad \mathbf{unsat} \\
& \Leftrightarrow d \geq 0 \wedge (\forall i. 0 \leq i < d \Rightarrow s[i] \geq 1) \wedge (0 \leq j < d \wedge s[j] < 0) \quad \mathbf{unsat} \\
& \Leftrightarrow d \geq 0 \wedge (\bigwedge_{i \in \{0, d-1, j\}} 0 \leq i < d \Rightarrow s[i] \geq 1) \wedge (0 \leq j < d \wedge s[j] < 0) \quad \mathbf{unsat}
\end{aligned}$$

In general, a vector judgment  $\Gamma \bar{\models} i \text{ in } ir$  also contains the structural vector operations `take`, `drop`, and `++`. These cannot be translated into the array property fragment, as they establish constraints between vector elements with different indices. For example, for vectors  $x :: \text{idxvec}(n), y :: \text{idxvec}(n+5)$  the property  $x \text{ in } \text{drop}(5, y)$  would translate to  $(\forall i. 0 \leq i < n \Rightarrow x[i] = y[i+5])$ . Unfortunately, it was shown in [16] that extending the array property fragment with arithmetic expressions over universally quantified index variables yields a fragment for which satisfiability is undecidable.

Nonetheless, almost all vector judgments arising in practical programs can still be decided, because the structural operations can be eliminated in a simple, yet effective preprocessing step. Only when the structural operations cannot be eliminated, the judgment can neither be validated nor refuted. In this situation, the program is rejected with an appropriate error message. We informally sketch out the transformation of judgments with structural vector operations by means of an example. The example arises during type checking of the generalized selection `gsel`.

$$\begin{aligned}
& \text{gsel} : \Pi r :: \text{nat}. \Pi s :: \text{natvec}(r). \\
& \quad \Pi l :: \{\text{nat in } .. r + 1\}. \Pi v :: \{\text{natvec}(l) \text{ in } .. \text{take}(l, s)\}. \\
& \quad [\text{int} | s] \rightarrow \text{numvec}(v) \rightarrow [\text{int} | \text{drop}(l, s)] \\
& \text{gsel } r' s' l' v \text{ a } x = \text{gen } y < \text{drop } \{\text{length } x, \text{shape } a\} \text{ of } [| | [0] |] \\
& \quad \text{with } a. [x ++ y]
\end{aligned}$$

In order to verify that the selection inside the `WITH`-loop does not exceed the array bounds, the following judgment must be validated.

$$\begin{aligned}
& r :: \text{idx}, r \text{ in } 0.., l :: \text{idx}, l \text{ in } 0..r+1, \\
& s :: \text{idxvec}(r), s \text{ in } \text{vec}(r, 0) \dots, v :: \text{idxvec}(l), v \text{ in } \text{vec}(l, 0) \dots \text{take}(l, s), \\
& y :: \text{idxvec}(r-l), y \text{ in } \text{vec}(r-l, 0) \dots \text{drop}(l, s) \bar{\models} v ++ y \text{ in } \text{vec}(r, 0) \dots
\end{aligned}$$

Vector  $v$  is constrained by the first  $l$  elements of  $s$  whereas  $y$  depends on the last  $r-l$  elements of  $s$ . Furthermore, the concatenation of  $v$  and  $y$  is compared to the entire vector  $s$ . During preprocessing,  $s$  is thus split into two vectors  $s_1$  of length  $l$  and  $s_2$  of length  $r-l$ . All occurrences of `take`( $l, s$ ) and `drop`( $l, s$ ) are then substituted with  $s_1$  and  $s_2$ , respectively.  $s$  itself is consistently replaced with  $s_1 ++ s_2$ .

$$\begin{aligned}
& r :: \text{idx}, r \text{ in } 0.., l :: \text{idx}, l \text{ in } 0..r+1, \\
& s_1 :: \text{idxvec}(l), s_2 :: \text{idxvec}(r-l), s_1 ++ s_2 \text{ in } \text{vec}(r, 0) \dots, \\
& v :: \text{idxvec}(l), v \text{ in } \text{vec}(l, 0) \dots s_1, y :: \text{idxvec}(r-l), y \text{ in } \text{vec}(r-l, 0) \dots s_2 \\
& \bar{\models} v ++ y \text{ in } \text{vec}(r, 0) \dots s_1 ++ s_2
\end{aligned}$$

The intermediate result has no `take` and `drop` operations left, but some concatenations. These are eliminated by splitting up the properties they appear in.  $s_1 ++ s_2 \text{ in } \text{vec}(r, 0) \dots$  is split into the two properties  $s_1 \text{ in } \text{vec}(l, 0) \dots, s_2 \text{ in } \text{vec}(r-l, 0) \dots$ . The conclusion  $v ++ y \text{ in } \text{vec}(r, 0) \dots s_1 ++ s_2$  is treated similarly. Both vectors  $v$  and  $s_1$  have length  $l$ . The property is thus split at that point, yielding the two properties  $v \text{ in } \text{vec}(l, 0) \dots s_1, y \text{ in } \text{vec}(r-l, 0) \dots s_2$ . The result contains no further structural operations. It may be validated after rewriting it as a formula in the array property fragment.

$$\begin{aligned}
& r :: \text{idx}, r \text{ in } 0.., l :: \text{idx}, l \text{ in } 0..r+1, \\
& s_1 :: \text{idxvec}(l), s_2 :: \text{idxvec}(r-l), s_1 \text{ in } \text{vec}(l, 0) \dots, s_2 \text{ in } \text{vec}(r-l, 0) \dots, \\
& v :: \text{idxvec}(l), v \text{ in } \text{vec}(l, 0) \dots s_1, y :: \text{idxvec}(r-l), y \text{ in } \text{vec}(r-l, 0) \dots s_2 \\
& \bar{\models} v \text{ in } \text{vec}(l, 0) \dots s_1, y \text{ in } \text{vec}(r-l, 0) \dots s_2
\end{aligned}$$

Elimination of structural operations fails if the constraints do not imply how to split a variable or a vector constraint into segments. We obtain an example of this when we change the order of  $x$  and  $y$  inside the selection of `gsel` and once more check whether all accesses to  $a$  are in bounds.

$$\begin{aligned}
& \text{gsel } r' s' l' v \text{ a } x = \text{gen } y < \text{drop } \{\text{length } x, \text{shape } a\} \text{ of } [| | [0] |] \\
& \quad \text{with } a. [y ++ x]
\end{aligned}$$

After eliminating `take` and `drop` operations as in the previous example, we get the following intermediate judgment.

$$\begin{aligned}
& r :: \text{idx}, r \text{ in } 0.., l :: \text{idx}, l \text{ in } 0..r+1, \\
& s_1 :: \text{idxvec}(l), s_2 :: \text{idxvec}(r-l), s_1 ++ s_2 \text{ in } \text{vec}(r, 0) \dots, \\
& v :: \text{idxvec}(l), v \text{ in } \text{vec}(l, 0) \dots s_1, y :: \text{idxvec}(r-l), y \text{ in } \text{vec}(r-l, 0) \dots s_2 \\
& \bar{\models} y ++ v \text{ in } \text{vec}(r, 0) \dots s_1 ++ s_2
\end{aligned}$$

In the property  $y ++ v \text{ in } \text{vec}(r, 0) \dots s_1 ++ s_2$ , the vectors  $y$  and  $s_1$  have length  $r-l$  and length  $l$ , respectively. The scalar constraints do not allow to derive whether  $r-l < l, r-l = l, \text{ or } r-l > l$  and thus the property cannot be split any further. In consequence, the entire program is rejected with a message that points out the location of the structural error.

Due to permuting  $x$  and  $y$ , the last variant of `gse1` was erroneous to start with and should not have been accepted anyways. In fact, we did not yet encounter a valid program that was rejected because of a structural problem. This is not surprising as the structure of shape vectors and array index vectors is crucial for every rank-generic program.

A potential alternative would be to rule out all cases in which the structural operations cannot be eliminated a priori by reflecting the structure of index vectors in their sort. For example, an index vector  $v_1$  could have the sort `idxvec( $l_1, l_2$ )` to indicate that it consists of two segments of the stated lengths. Whenever it is combined with other vectors  $v_2, v_3$  in a dyadic operation  $f(v_1, v_2)$  or in a vector property  $v_1 \text{ in } v_2 \dots v_3$ , the other vectors must have provably the same structure. By construction, all structural operations could then be eliminated in single step, allowing to rewrite the judgment in the array property fragment immediately.

## 7. Related Work

The work presented in this paper combines multidimensional, irregularly nested array programming with dependent types. In the following, we briefly mention work from the different areas of programming language research that's related to our's.

Array languages like MATLAB [5], APL [3,17], J [4] or NIAL [18] are interpreted and mostly untyped. In particular they are known for offering a plethora of well optimized operators for each array operation supported by the language. This stands in contrast to our work in which we try to condense the essence of multidimensional array programming into a small number of primitively recursive constructs.

As soon as attempts are made to compile array programs for efficient execution, knowledge about the array properties and their relationships becomes crucial. For example in FISH [19], each function  $f$  is accompanied by a shape function  $\#f$  which maps the shape of the argument to the shape of the result. Shape inference proceeds by first inlining all functions and then statically evaluating all shape functions. FISH rejects all programs that contain non-constant array shapes. In our approach, we may statically verify shape- and rank-generic programs without excessive inlining. Rediscovering array properties for better compilation of untyped array languages such as MATLAB is an area of ongoing research, see for example [20–22]. In our context the array types contain everything the programmer knows about the structural properties of the program, eliminating the need for such work.

The field of functional array programming was pioneered by SISAL [23] and NESL [24]. SISAL demonstrated that functional array programming and implicit parallelization can achieve competitive run time performance, despite the aggregate update problem. While SISAL restricts itself to (one-dimensional) vectors of homogeneously nested vectors, NESL also supports irregularly nested vectors. Recent work has been going on to integrate nested data-parallelism into HASKELL [25,26]. In contrast to our work, these approaches provide no support for truly multidimensional arrays.

As the last field of related work we survey the research area of dependently typed programming [27]. Dependent types naturally lend themselves for describing arrays as they allow the use of (dynamic) terms to index within families of types. Indeed, the classical example for dependently typed programming is the index family of vectors from which an element with a particular length is selected. The expressive power of dependent types renders the problem of type equality generally undecidable as it boils down to deciding whether any two expressions denote the same value. For example, CAYENNE [28] is a fully dependently typed language. Its type system is undecidable and it lacks phase distinction. Both problems can be overcome by restricting the type language as done in EPIGRAM [29,30], which rules out general recursion in type-forming expressions to retain decidability. Recently, the YNOT project aims at integrating dependent types into programming systems with effectful computations [31].

Most closely related to our approach are more light-weight approaches such as Xi and Pfenning's DML [11], Xi's *applied type system* [32], and Zenger's *indexed types* [10]. These approaches allow term-indexing into type families only for certain *index sorts*. The type-checking problem is reduced to constraint solving on these sorts, which is decidable. Our work shares some of its technical underpinnings with DML. Xi and Pfenning also proposed the use of dependent types for the elimination of array boundary checks. However, apart from that, DML offered no particular support for array programming or data parallelism.

## 8. Conclusion

Making the expressive power of dependent types available for practical program development is a subject of ongoing research. It is a particular challenge to design programming systems with dependent types in a way such that a user is not required to have expert knowledge in type theory. We think that in the array programming paradigm, employing dependent types is both intuitive and beneficial.

Dependent types are intuitive for array programs because rank and shape are inherent properties of multidimensional arrays. Scientific programmers are used to specifying their algorithms in terms of array shapes: every undergraduate course on linear algebra teaches the type of matrix multiplication as  $\mathbb{R}^{m \times n} \times \mathbb{R}^{n \times p} \rightarrow \mathbb{R}^{m \times p}$ . For specifications like this, dependent types allow the developer to concisely express the function signature in a computer program.

Dependent types are beneficial for array programs, because structural constraints are crucial for their safe evaluation. A type system with dependent types can statically enforce the relevant constraints, thus ruling out programs that may fail during evaluation. Without potential run time errors, the accepted programs do not need to perform expensive run time

checks. Moreover, a compiler can exploit the structural properties encoded in the dependent types for extensive program optimization.

Since our type system uses an SMT solver to verify the necessary constraints, type checking proceeds fully automatically. The system thus resembles a type system for a mainstream programming language that either accepts or rejects a program with an appropriate message. In case of rejecting a program, our system can even provide precise values of the index variables for which the program will fail. This behavior is similar to a model checking tool that yields a counter example for which the desired property is violated.

The ideas presented in this paper form the basis of the functional array programming language Qube. We are currently developing a compiler [33] for Qube that implements dependent array types as proposed in this paper. To simplify programming with indexed types, the system allows implicit index arguments which are automatically reconstructed if omitted [34]. We envision to exploit the information provided by the dependent types to generate more efficient array programs both for sequential and parallel execution. For example, we may replace selections into arrays defined by means of WITH-loops with the selected element's definition, thereby achieving deforestation. Furthermore, in combination with a memory management scheme based on run time reference counting, we may often perform destructive array updates even in our context of immutable arrays. The SAC compiler performs similar optimizations [35,36], but to preserve program semantics, they are only applicable if array shapes are compile-time constants. In Qube, dependent array types will allow us to overcome this restriction. Eventually, a future version of SAC may incorporate the essential concepts of Qube.

## Acknowledgments

We would like thank the anonymous referees for their constructive criticism. We also thank Johannes Blume, Florian Büther, and Markus Weigel for contributing to the Qube project. Last but not least, we thank Walter Dosch for our interesting discussions and his helpful comments on the manuscript.

## References

- [1] H. Sutter, A fundamental turn toward concurrency in software, *Dr. Dobbs's J.*, 30 (3) (2005).
- [2] C. Grelck, Shared memory multiprocessor support for functional array processing in SAC, *J. Funct. Programming* 15 (3) (2005) 353–401.
- [3] K. Iverson, *A Programming Language*, John Wiley, New York City, NY, USA, 1962.
- [4] K. Iverson, *J Introduction and Dictionary*, Iverson Software Inc., Toronto, Canada, 1995.
- [5] C. Moler, J. Little, S. Bangert, *Pro-Matlab User's Guide*, The MathWorks, Cochituate Place, 24 Prime Park Way, Natick, MA, USA, 1987.
- [6] C. Grelck, S.-B. Scholz, SAC: a functional array language for efficient multithreaded execution, *Internat. J. Parallel Programming* 34 (4) (2006) 383–427.
- [7] C. Grelck, S.-B. Scholz, A. Shafarenko, A binding scope analysis for generic programs on arrays, in: A. Butterfield (Ed.), *Implementation and Application of Functional Languages*, 17th International Workshop (IFL'05), Dublin, Ireland, September 19–21, 2005, Revised Selected Papers, Lecture Notes in Computer Science, vol. 4015, Springer-Verlag, Berlin, Heidelberg, New York, 2006, pp. 212–230.
- [8] S. Herhut, S.-B. Scholz, R. Bernecky, C. Grelck, K. Trojahner, From contracts towards dependent types: proofs by partial evaluation, in: O. Chitil, Z. Horváth, V. Zsóok (Eds.), *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages (IFL'07)*, Freiburg, Germany, Revised Selected Papers, Lecture Notes in Computer Science, vol. 5083, Springer-Verlag, Berlin, Heidelberg, New York, 2008, pp. 254–273.
- [9] H. Xi, F. Pfenning, Eliminating array bound checking through dependent types, in: *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, 1998, pp. 249–257.
- [10] C. Zenger, Indexed types, *Theoret. Comput. Sci.* 187 (1–2) (1997) 147–165.
- [11] H. Xi, F. Pfenning, Dependent types in practical programming, in: A. Aiken (Ed.), *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, San Antonio, Texas, USA, ACM Press, 1999, pp. 214–227.
- [12] B. Pierce, *Types and Programming Languages*, MIT Press, Cambridge, MA, USA, 2002.
- [13] K. Trojahner, C. Grelck, Dependently typed array programs don't go wrong, *Tech. Rep. A-08-06*, Schriftenreihe der Institute für Informatik/Mathematik, University of Lübeck, Lübeck, Germany, 2008. <<http://www.isp.uni-luebeck.de/papers/2008/dtapdgv-tr08.pdf>>.
- [14] B. Dutertre, L. de Moura, A fast linear-arithmetic solver for DPLL(T), in: T. Ball, R.B. Jones (Eds.), *Computer Aided Verification*, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17–20, 2006, Proceedings, Lecture Notes in Computer Science, vol. 4144, Springer Verlag, Berlin, Heidelberg, 2006, pp. 81–94.
- [15] L. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: C. Ramakrishnan, J. Rehof (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29–April 6, 2008, Proceedings, Lecture Notes in Computer Science, vol. 4963, Springer Verlag, Berlin, Heidelberg, 2008, pp. 337–340.
- [16] A.R. Bradley, Z. Manna, H.B. Sipma, What's decidable about arrays?, in: E.A. Emerson, K.S. Namjoshi (Eds.), *Verification, Model Checking, and Abstract Interpretation*, Seventh International Conference, VMCAI 2006, Charleston, SC, USA, January 8–10, 2006, Proceedings, Lecture Notes in Computer Science, vol. 3855, Springer Verlag, Berlin, Heidelberg, 2006, pp. 427–442.
- [17] A. Falkoff, K. Iverson, The design of APL, *IBM J. Res. Develop.* 17 (4) (1973) 324–334.
- [18] M. Jenkins, Q'Nial: a portable interpreter for the nested interactive array language, *Nial*, *Software Pract. Exp.* 19 (2) (1989) 111–126.
- [19] C. Jay, P. Steckler, The functional imperative: shape!, in: C. Hankin (Ed.), *Proceedings of the Seventh European Symposium on Programming (ESOP'98)*, Part of the Joint European Conference on Theory and Practice of Software (ETAPS'98), Lisbon, Portugal, Lecture Notes in Computer Science, vol. 1381, Springer Verlag, Berlin, Germany, 1998, pp. 139–153.
- [20] L. de Rose, D. Padua, Techniques for the translation of MATLAB programs into Fortran 90, *ACM Trans. Programming Lang. Syst.* 21 (2) (1999) 286–323.
- [21] C. McCosh, Type-based specialization in a telescoping compiler for MATLAB, Master Thesis TR03-412, Rice University, Houston, Texas, USA, 2003.
- [22] P. Joisha, P. Banerjee, An algebraic array shape inference system for MATLAB, *ACM Trans. Programming Lang. Syst.* 28 (5) (2006) 848–907.
- [23] D. Cann, Retire Fortran? A debate rekindled, *Comm. ACM* 35 (8) (1992) 1–89.
- [24] G. Blelloch, Programming parallel algorithms, *Comm. ACM* 39 (3) (1996) 85–97.
- [25] M.M. Chakravarty, G. Keller, An approach to fast arrays in Haskell, in: J. Jeuring, S.P. Jones (Eds.), *Summer School and Workshop on Advanced Functional Programming*, Oxford, England, UK, 2002, Lecture Notes in Computer Science, vol. 2638, Springer Verlag, Berlin, Germany, 2003, pp. 27–58.
- [26] M.M.T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, S. Marlow, Data parallel Haskell: a status report, in: *DAMP '07: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, ACM Press, New York, NY, USA, 2007, pp. 10–18.
- [27] J. Smith, B. Nordström, K. Petersson, *Programming in Martin-Löf's Type Theory*, Oxford University Press, 1990.

- [28] L. Augustsson, Cayenne – a language with dependent types, International Conference on Functional Programming, ACM, New York, NY, USA, 1998, pp. 239–250.
- [29] C. McBride, J. McKinna, The view from the left, *J. Funct. Programming* 14 (1) (2004) 69–111.
- [30] T. Altenkirch, C. McBride, J. McKinna, Why dependent types matter (April 2005). <<http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf>>.
- [31] A. Nanevski, G. Morrisett, L. Birkedal, Polymorphism and separation in Hoare type theory, International Conference on Functional Programming, ACM Press, New York, NY, USA, 2006, pp. 62–73.
- [32] H. Xi, Applied type system (extended abstract), in: S. Baradi, M. Coppo, F. Damiani (Eds.), Types for Proofs and Programs, Third International Workshop, TYPES 2003, Torino, Italy, April 30–May 4, 2003, Revised Selected Papers, Lecture Notes in Computer Science, vol. 3085, Springer Verlag, Berlin, Germany, 2004, pp. 394–408.
- [33] F. Büther, A Compiler for Qube, A Functional Array Programming Language with Dependent Types, Studienarbeit, Institute for Software Technology and Programming Languages, University of Lübeck, Lübeck, Germany, 2008.
- [34] M. Weigel, Facilitating array programming with dependent types – precise error messages and implicit index arguments for Qube, Master’s thesis, Institute for Software Technology and Programming Languages, University of Lübeck, Lübeck, Germany, 2008.
- [35] S.-B. Scholz, WITH-loop-folding: condensing consecutive array operations, in: C. Clack, T. Davie, K. Hammond (Eds.), Proceedings of the Ninth International Workshop on Implementation of Functional Languages (IFL’97), St. Andrews, Scotland, UK, University of St. Andrews, 1997, pp. 225–242.
- [36] C. Grelck, K. Trojahnner, Implicit memory management for SAC, in: C. Grelck, F. Huch (Eds.), Proceedings of the 16th International Workshop on Implementation and Application of Functional Languages, IFL 2004, Lübeck, Germany, September 8–10, 2004, Technical Report, vol. 0408, University of Kiel, 2004, pp. 335–348.