Dependently Typed Array Programs Don't Go Wrong

Kai Trojahner [*], Clemens Grelck

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

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 of such violations 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.

In this paper, 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.

Key words: Array Programming, Dependent Types, Program Verification

1 Introduction

In the array programming paradigm multidimensional arrays serve as the fundamental data structures of computation. Such arrays can be vectors, matrices,

[*] Corresponding author.
Email addresses: `trojahner@isp.uni-luebeck.de` (Kai Trojahner), `grelck@isp.uni-luebeck.de` (Clemens Grelck).

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 multicore processors [?,?]. Prominent examples of array languages are APL [?], J [?], MatLab [?], and SaC [?].

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 [?]. SaC has a static type system that employs three layers of array types. While the array element type is always monomorphic, 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 [?] and partial evaluation techniques [?] 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 [?,?]. 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:

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

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 [?], 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 important 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 pro-

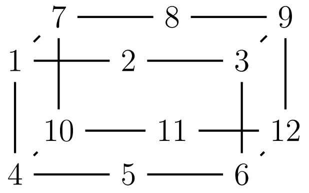
Array	Rank	Shape vector
1	0	[]
$\lceil 1\ 2\ 3 \rceil$	1	[3]
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	2	[2 3]
	3	[2 2 3]

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

cessed 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 truely 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

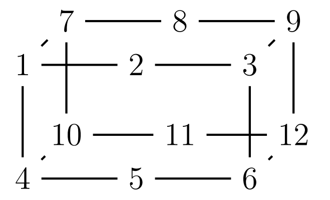
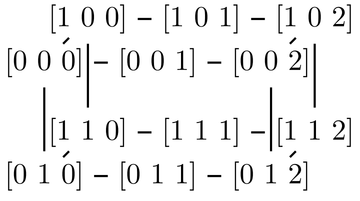
Array	**Index vectors**
1	[]
$\boxed{1\ 2\ 3}$	[[0] [1] [2]]
$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}$	$\begin{pmatrix}[0\ 0] & [0\ 1] & [0\ 2]\\ [1\ 0] & [1\ 1] & [1\ 2]\end{pmatrix}$
	 [1 0 0] – [1 0 1] – [1 0 2] [0 0 0] – [0 0 1] – [0 0 2] [1 1 0] – [1 1 1] – [1 1 2] [0 1 0] – [0 1 1] –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

Fig. 2. Example arrays and the legal index vectors

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

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

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 $a_1, ..., a_n$. 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

Syntactic forms

$$\begin{array}{lll}
t ::= [|\, q^p \,|\, [s^d] \,|] \mid \texttt{rank}\; t \mid \texttt{shape}\; t \mid \texttt{sel}(t,t) & & \text{Terms} \\
q ::= c & & \text{Quarks} \\
v ::= [|\, q^p \,|\, [s^d] \,|] & & \text{Values}
\end{array}$$

Evaluation rules

$$\frac{t \longrightarrow t'}{\texttt{rank}\; t \longrightarrow \texttt{rank}\; t'} \qquad \texttt{rank}\; [|\, q^p \,|\, [s^d] \,|] \longrightarrow [|\, d \,|\, [\,] \,|]$$

$$\frac{t \longrightarrow t'}{\texttt{shape}\; t \longrightarrow \texttt{shape}\; t'} \qquad \texttt{shape}\; [|\, q^p \,|\, [s^d] \,|] \longrightarrow [|\, s^d \,|\, [d] \,|]$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{sel}(t_1,t_2) \longrightarrow \texttt{sel}(t_1',t_2)} \qquad \frac{t_2 \longrightarrow t_2'}{\texttt{sel}(v_1,t_2) \longrightarrow \texttt{sel}(v_1,t_2')}$$

$$\frac{\forall k.\, 0 \leq i_k < s_k}{\texttt{sel}([|\, q^p \,|\, [s^d] \,|],\, [|\, i^d \,|\, [d] \,|]) \longrightarrow [|\, q_{\iota(d,s^d,i^d)} \,|\, [\,] \,|]}$$

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

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) $p = \Pi_{i=1}^{d} s_i$.

Inside the data vector, the elements along the innermost array axis are stored densely (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 ι then determines the linear index of the element at position v in the data vector of A:

$$\iota(r, s^r, v^r) = \Sigma_{i=1}^{r} \left(v_i \cdot \Pi_{j=i+1}^{r} s_j\right) + 1.$$

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

$$\texttt{rank}\; [|\, 1,2,3,4,5,6 \,|\, [2,3] \,|] \longrightarrow [|\, 2 \,|\, [\,] \,|]$$

$$\texttt{shape}\; [|\, 1,2,3,4,5,6 \,|\, [2,3] \,|] \longrightarrow [|\, 2,3 \,|\, [2] \,|]$$

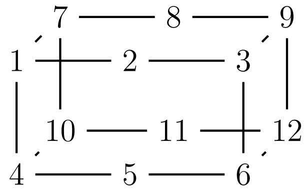
<table>
<tr><th>Array</th><th>Uniform array representation</th></tr>
<tr><td>1</td><td>[| 1 | [] |]</td></tr>
<tr><td>$\boxed{1\ 2\ 3}$</td><td>[| 1, 2, 3 | [3] |]</td></tr>
<tr><td>$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$</td><td>[| 1, 2, 3, 4, 5, 6 | [2, 3] |]</td></tr>
<tr><td> </td><td>[| 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 | [2, 2, 3] |]</td></tr>
</table>

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.

$$\texttt{shape}\ (\texttt{shape}\ \texttt{[|}\,q^n\,\texttt{|}\,\texttt{[}s^d\texttt{]}\,\texttt{|]})\ \longrightarrow^*\ \texttt{[|}\,d\,\texttt{|}\,\texttt{[1]}\,\texttt{|]}$$

$$\texttt{rank}\ \texttt{[|}\,q^n\,\texttt{|}\,\texttt{[}s^d\texttt{]}\,\texttt{|]}\ \longrightarrow\ \texttt{[|}\,d\,\texttt{|}\,\texttt{[]}\,\texttt{|]}$$

A selection $\texttt{sel(}A\texttt{,}\,\texttt{[|}\,i^e\,\texttt{|}\,\texttt{[}e\texttt{]}\,\texttt{|]}\texttt{)}$ into a multidimensional array $A = \texttt{[|}\,q^p\,\texttt{|}\,\texttt{[}s^d\texttt{]}\,\texttt{|]}$ 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 $[3, 3]$:

$$\frac{\overline{0 \le 1 < 3\ \wedge\ 0 \le 1 < 3}}{\texttt{sel(}\texttt{[|}\,1,2,3,4,5,6,7,8,9\,\texttt{|}\,\texttt{[3,3]}\,\texttt{|]}\,\texttt{,}\,\texttt{[|}\,1,1\,\texttt{|}\,\texttt{[2]}\,\texttt{|]}\texttt{)}\ \longrightarrow\ \texttt{[|}\,5\,\texttt{|}\,\texttt{[]}\,\texttt{|]}}$$

The evaluation rules for both rank and shape are straightforward: Whenever the argument reduces to 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

In this section, we specify a core language that captures the essential features necessary for functional array programming. The 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 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. To keep type checking decidable, 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

As mentioned before, types may only depend on the terms of a specific index language in order to keep type checking decidable. The index terms are solely used for type checking; they are not subject to evaluation. All index terms belong to an index sort. `idx` is the sort of integer scalars, `idxvec(`i`)` is the sort-family of integer vectors. In this sort family, a sort for vectors of a particular

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

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

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 `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 $\texttt{[}0,1,2\texttt{]}$ belongs to the sort $\texttt{idxvec(}3\texttt{)}$. 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, $\texttt{vec(}l\texttt{,}i\texttt{)}$ 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 $a \mathbin{+\!\!+} b$ which appends the vector b of length l_b to the vector a of length l_a. Naturally, the result is of sort $\texttt{idxvec(}l_a \texttt{+} l_b\texttt{)}$. Conversely, vectors can be split using the operations `take` and `drop`. For a given vector v of length l and a scalar index expression i with $0 \leq i \leq l$, $\texttt{take(}i\texttt{,}v\texttt{)}$ and $\texttt{drop(}i\texttt{,}v\texttt{)}$ denote the prefix of v with length i and the suffix of v with length $l \texttt{-} i$, respectively. Thus we have

$\texttt{take}(i,v) \mathbin{+\!\!+} \texttt{drop}(i,v) = v$.

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

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 $\texttt{[}Q\,|\,i\texttt{]}$ 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 $\texttt{[|}1,2,3,4\,|\,\texttt{[}4\texttt{]}\texttt{|]}$ is $\texttt{[int}\,|\,\texttt{[}4\texttt{]]}$, while a scalar integer $\texttt{[|}7\,|\,\texttt{[]}\texttt{|]}$ has type $\texttt{[int}\,|\,\texttt{[]]}$.

The integer quarks of type $\texttt{int}$ 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 $T_1 \to T_2$, index abstractions $\lambda' x :: I.\, t$ of type $\Pi x :: I.\, T$, tuples of arrays values $\{v_1, .., v_n\}$ of type $\{T_1, .., T_n\}$, and dependent pairs $\{'i, v : \Sigma x :: I.\, T\}$ of type $\Sigma x :: I.\, T$. The bottom quark type $\bot_Q$ is not associated with a particular quark. Instead, it serves as a quark type for empty arrays such as the empty vector $\texttt{[|}\,|\,\texttt{[}0\texttt{]}\texttt{|]}$ which has type $\texttt{[}\bot_Q\,|\,\texttt{[}0\texttt{]]}$. To capture the intuition that an empty array may have an arbitrary quark type, $\bot_Q$ 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 $\texttt{num}(i)$ characterizes all scalar integer arrays whose quark is identical to the index i. By means of subtyping, each $\texttt{num}(i)$ is also an $\texttt{[int}\,|\,\texttt{[]]}$. Similarly, an integer vector of type $\texttt{numvec}(i)$ is also an $\texttt{[int}\,|\,\texttt{[}l\texttt{]]}$ provided that the index vector i is of sort $\texttt{idxvec}(l)$. Thus, the above arrays $\texttt{[|}7\,|\,\texttt{[]}\texttt{|]}$ and $\texttt{[|}1,2,3,4\,|\,\texttt{[}4\texttt{]}\texttt{|]}$ also have the more specific types $\texttt{num(7)}$ and $\texttt{numvec(}[1,2,3,4]\texttt{)}$, respectively.

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ (\text{E-App1}) \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ (\text{E-App2})$$

$$\texttt{[|}\,\lambda x : T.\, t\,\texttt{|}\ \texttt{[]}\ \texttt{|]}\ v_2 \longrightarrow t[x \mapsto v_2]\ (\text{E-AppAbs})$$

$$\frac{t \longrightarrow t'}{t\ 'i \longrightarrow t'\ 'i}\ (\text{E-IApp})$$

$$\texttt{[|}\,\lambda' x :: I.\, t\,\texttt{|}\ \texttt{[]}\ \texttt{|]}\ 'i \longrightarrow t[x \mapsto_i i]\ (\text{E-IAppIAbs})$$

$$\frac{t_j \longrightarrow t_j'}{\{v^{j-1}, t_j, t^{n-j}\} \longrightarrow \{v^{j-1}, t_j', t^{n-j}\}}\ (\text{E-Tup1})$$

$$\{v^n\} \longrightarrow \texttt{[|}\,\{v^n\}\,\texttt{|}\ \texttt{[]}\ \texttt{|]}\ (\text{E-Tup2})$$

$$\frac{t \longrightarrow t'}{\{'i\,\texttt{,}t\ \texttt{:}\ \Sigma x :: I.\, T\} \longrightarrow \{'i\,\texttt{,}t'\ \texttt{:}\ \Sigma x :: I.\, T\}}\ (\text{E-ITup1})$$

$$\{'i\,\texttt{,}v\ \texttt{:}\ \Sigma x :: I.\, T\} \longrightarrow \texttt{[|}\,\{'i\,\texttt{,}v\texttt{:}\Sigma x :: I.\, T\}\,\texttt{|}\ \texttt{[]}\ \texttt{|]}\ (\text{E-ITup2})$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{let}\ p\ \texttt{=}\ t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ p\ \texttt{=}\ t_1'\ \texttt{in}\ t_2}\ (\text{E-Let})$$

$$\texttt{let}\ x\ \texttt{=}\ v_1\ \texttt{in}\ t_2 \longrightarrow t_2[x \mapsto v_1]\ (\text{E-LetVal})$$

$$\texttt{let}\ \{x^i\}\ \texttt{=}\ \texttt{[|}\,\{v^i\}\,\texttt{|}\ \texttt{[]}\ \texttt{|]}\ \texttt{in}\ t_2 \longrightarrow t_2[x_1 \mapsto v_1]..[x_n \mapsto v_n]\ (\text{E-LetTup})$$

$$\texttt{let}\ \{'x_1, x_2\}\ \texttt{=}\ \texttt{[|}\,\{'i\,\texttt{,}v\texttt{:}\Sigma x :: I.\, T\}\,\texttt{|}\ \texttt{[]}\ \texttt{|]}\ \texttt{in}\ t_2 \longrightarrow$$
$$t_2[x_1 \mapsto_i i][x_2 \mapsto v]\ (\text{E-LetITup})$$

Fig. 6. Basic semantics of typed array programs

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 is defined in Fig. 6.

3.3.1 Functions

The abstraction quark $\lambda x : T_1.\, t$ allows to specify arrays of functions. Its type is the function quark type $T_1 \to T_2$. 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 $\texttt{[|}\,\lambda x : T.\, t\,\texttt{|}\ \texttt{[]}\ \texttt{|]}$, the entire application will take a β-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 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 $\texttt{[|}\, \lambda' x :: I.\, t \,\texttt{|}\, \texttt{[]} \,\texttt{|]}$, the index application takes an evaluation step by substituting all index identifiers x in t with i.

3.3.2 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 $\{v_1, ..., v_n\}$ of type $\{T_1, ..., T_n\}$ 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 $\{'i \texttt{,} v \texttt{:} \Sigma x :: I.\, T\}$ 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 $\{'2,\ \texttt{[|}\, 2, 2 \,\texttt{|}\, \texttt{[}2\texttt{]} \,\texttt{|]}\}$ has type $\Sigma x :: \texttt{nat}.\ \texttt{[int|}\,\texttt{[}x\texttt{]]}$, but also the types $\Sigma x :: \texttt{nat}.\ \texttt{[int|}\,\texttt{[}2\texttt{]]}$, $\Sigma x :: \texttt{nat}.\, \texttt{numvec}(\texttt{[}x, x\texttt{]})$, and $\Sigma x :: \texttt{nat}.\, \texttt{numvec}(\texttt{[}2, x\texttt{]})$, among others. Vice versa, several dependent pairs have the same type: both dependent tuples $\{'2,\ \texttt{[|}\, 2, 2 \,\texttt{|}\, \texttt{[}2\texttt{]} \,\texttt{|]}\}$ and $\{'3,\ \texttt{[|}\, 1, 2, 3 \,\texttt{|}\, \texttt{[}3\texttt{]} \,\texttt{|]}\}$ have the type $\Sigma x :: \texttt{nat}.\ \texttt{[int|}\,\texttt{[}x\texttt{]]}$. 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 $\{t_1, ..., t_n\}$ 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 $\{v_1, .., v_n\}$ according to rule E-Tup2. Analogously, there is also constructor term for dependent pairs $\{'i \texttt{,} t : \Sigma x :: I.\, T\}$ which is explained by the rules E-ITup1 and E-ITup2.

$$\frac{t \longrightarrow t'}{f\ t \longrightarrow f\ t'}\ (\text{E-PrfApp})$$

$$\texttt{rank}\ \texttt{[|}\, q^p \,\texttt{|}\, \texttt{[}s^d\texttt{]} \,\texttt{|]} \ \longrightarrow\ \texttt{[|}\, d \,\texttt{|}\, \texttt{[]} \,\texttt{|]}\ (\text{E-Rank})$$

$$\texttt{shape}\ \texttt{[|}\, q^p \,\texttt{|}\, \texttt{[}s^d\texttt{]} \,\texttt{|]} \ \longrightarrow\ \texttt{[|}\, s^d \,\texttt{|}\, \texttt{[}d\texttt{]} \,\texttt{|]}\ (\text{E-Shape})$$

$$\texttt{length}\ \texttt{[|}\, q^l \,\texttt{|}\, \texttt{[}l\texttt{]} \,\texttt{|]} \ \longrightarrow\ \texttt{[|}\, l \,\texttt{|}\, \texttt{[]} \,\texttt{|]}\ (\text{E-Length})$$

$$\begin{gathered} f_2\ \ \texttt{[|}\,\{\,\texttt{[|}\, q^p \,\texttt{|}\, \texttt{[}s^d\texttt{]} \,\texttt{|]},\, \texttt{[|}\, r^p \,\texttt{|}\, \texttt{[}s^d\texttt{]} \,\texttt{|]}\,\} \,\texttt{|}\, \texttt{[]} \,\texttt{|]} \ \longrightarrow \\ \texttt{[|}\, \tilde{f_2}(q_1, r_1), .., \tilde{f_2}(q_p, r_p) \,\texttt{|}\, \texttt{[}s^d\texttt{]} \,\texttt{|]}\ (\text{E-Bin}) \end{gathered}$$

$$\frac{\forall k.\, 0 \le i_k < s_i}{\texttt{sel}\,\texttt{[|}\,\{\,\texttt{[|}\, q^p \,\texttt{|}\, \texttt{[}s^d\texttt{]} \,\texttt{|]},\, \texttt{[|}\, i^d \,\texttt{|}\, \texttt{[}d\texttt{]} \,\texttt{|]}\,\} \,\texttt{|}\, \texttt{[]} \,\texttt{|]} \ \longrightarrow\ \texttt{[|}\, q_{\iota(d,s^d,i^d)} \,\texttt{|}\, \texttt{[]} \,\texttt{|]}}\ (\text{E-Sel})$$

$$\frac{l \ge 0}{\texttt{vec}\,\texttt{[|}\,\{\,\texttt{[|}\, l \,\texttt{|}\, \texttt{[]} \,\texttt{|]},\, \texttt{[|}\, q \,\texttt{|}\, \texttt{[]} \,\texttt{|]}\,\} \,\texttt{|}\, \texttt{[]} \,\texttt{|]} \ \longrightarrow\ \texttt{[|}\, \underbrace{q, .., q}_{l} \,\texttt{|}\, \texttt{[}l\texttt{]} \,\texttt{|]}}\ (\text{E-Vec})$$

$$\texttt{++}\,\texttt{[|}\,\{\,\texttt{[|}\, q^m \,\texttt{|}\, \texttt{[}m\texttt{]} \,\texttt{|]},\, \texttt{[|}\, q'^n \,\texttt{|}\, \texttt{[}n\texttt{]} \,\texttt{|]}\,\} \,\texttt{|}\, \texttt{[]} \,\texttt{|]} \ \longrightarrow\ \texttt{[|}\, q^m, q'^n \,\texttt{|}\, \texttt{[}m \tilde{+} n\texttt{]} \,\texttt{|]}\ (\text{E-Cat})$$

$$\frac{0 \le n \le l}{\texttt{take}\,\texttt{[|}\,\{\,\texttt{[|}\, n \,\texttt{|}\, \texttt{[]} \,\texttt{|]},\, \texttt{[|}\, q^l \,\texttt{|}\, \texttt{[}l\texttt{]} \,\texttt{|]}\,\} \,\texttt{|}\, \texttt{[]} \,\texttt{|]} \ \longrightarrow\ \texttt{[|}\, q_1, .., q_n \,\texttt{|}\, \texttt{[}n\texttt{]} \,\texttt{|]}}\ (\text{E-Take})$$

$$\frac{0 \le n \le l}{\texttt{drop}\,\texttt{[|}\,\{\,\texttt{[|}\, n \,\texttt{|}\, \texttt{[]} \,\texttt{|]},\, \texttt{[|}\, q^l \,\texttt{|}\, \texttt{[}l\texttt{]} \,\texttt{|]}\,\} \,\texttt{|}\, \texttt{[]} \,\texttt{|]} \ \longrightarrow\ \texttt{[|}\, q_{n+1}, .., q_l \,\texttt{|}\, \texttt{[}l \tilde{-} n\texttt{]} \,\texttt{|]}}\ (\text{E-Drop})$$

$$\frac{t_j \longrightarrow t'_j}{\texttt{[}t^{j-1}, t_j, t^{n-j} \,\texttt{|}\, \texttt{[}f^d\texttt{]]} \ \longrightarrow\ \texttt{[}t^{j-1}, t'_j, t^{n-j} \,\texttt{|}\, \texttt{[}f^d\texttt{]]}}\ (\text{E-Arr1})$$

$$\texttt{[[|}\, q_i^p \,\texttt{|}\, \texttt{[}c^e\texttt{]} \,\texttt{|]}^n \,\texttt{|}\, \texttt{[}f^d\texttt{]]} \ \longrightarrow\ \texttt{[|}\, q_1^p, .., q_n^p \,\texttt{|}\, \texttt{[}f^d, c^e\texttt{]} \,\texttt{|]}\ (\text{E-Arr2})$$

$$\frac{t_1 \longrightarrow t'_1}{\texttt{gen}\ x \ \texttt{<}\ t_1\ \texttt{of}\ t_2\ \texttt{with}\ t_3 \longrightarrow \texttt{gen}\ x \ \texttt{<}\ t'_1\ \texttt{of}\ t_2\ \texttt{with}\ t_3}\ (\text{E-GenF})$$

$$\frac{t_2 \longrightarrow t'_2}{\texttt{gen}\ x \ \texttt{<}\ v_1\ \texttt{of}\ t_2\ \texttt{with}\ t_3 \longrightarrow \texttt{gen}\ x \ \texttt{<}\ v_1\ \texttt{of}\ t'_2\ \texttt{with}\ t_3}\ (\text{E-GenC})$$

$$\frac{\forall k.\, f_k \ge 0 \qquad \exists j.\, f_j = 0}{\texttt{gen}\ x \ \texttt{<}\ \texttt{[|}\, f^d \,\texttt{|}\, \texttt{[}d\texttt{]} \,\texttt{|]}\ \texttt{of}\ \texttt{[|}\, c^e \,\texttt{|}\, \texttt{[}e\texttt{]} \,\texttt{|]}\ \texttt{with}\ t \ \longrightarrow\ \texttt{[|}\,\texttt{|}\, \texttt{[}f^d, c^e\texttt{]} \,\texttt{|]}}\ (\text{E-GenE})$$

$$\frac{\forall k.\, f_k > 0 \qquad \forall y^d \in \vec{0}..f^d.\ c_{\iota(d,f^d,y^d)} = t[x \mapsto_i \texttt{[}y^d\texttt{]}][x \mapsto \texttt{[|}\, y^d \,\texttt{|}\, \texttt{[}d\texttt{]} \,\texttt{|]}]}{\texttt{gen}\ x \ \texttt{<}\ \texttt{[|}\, f^d \,\texttt{|}\, \texttt{[}d\texttt{]} \,\texttt{|]}\ \texttt{of}\ v\ \texttt{with}\ t \ \longrightarrow\ \texttt{[}c^p \,\texttt{|}\, \texttt{[}f^d\texttt{]]}}\ (\text{E-Gen})$$

$$\frac{t_1 \longrightarrow t'_1}{\texttt{loop}\ x_1 \ \texttt{<}\ t_1\,\texttt{,}\ x_2 \ \texttt{=}\ t_2\ \texttt{with}\ t_3 \longrightarrow \texttt{loop}\ x_1 \ \texttt{<}\ t'_1\,\texttt{,}\ x_2 \ \texttt{=}\ t_2\ \texttt{with}\ t_3}\ (\text{E-Loop1})$$

$$\frac{\begin{gathered}\forall k.\, 0 \le s_k \\ \forall v^d \in \vec{0}..s^d.\ f_{\iota(d,s^d,v^d)} = \texttt{[|}\, \lambda y.\, t_3[x \mapsto_i \texttt{[}v^d\texttt{]}][x \mapsto \texttt{[|}\, v^d \,\texttt{|}\, \texttt{[}d\texttt{]} \,\texttt{|]}] \,\texttt{|}\, \texttt{[]} \,\texttt{|]}\end{gathered}}{\texttt{loop}\ x \ \texttt{<}\ \texttt{[|}\, s^d \,\texttt{|}\, \texttt{[}d\texttt{]} \,\texttt{|]}\,\texttt{,}\ y \ \texttt{=}\ t_2\ \texttt{with}\ t_3 \ \longrightarrow\ f_p\ ...(f_1\ t_2)}\ (\text{E-Loop2})$$

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

3.3.3 Let binding

The `let` binding allows to give names to 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 $\{v_1, .., v_n\}$, the binding `let` $\{x_1, .., x_n\}$ `=` 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 $\{'i\texttt{,} v\texttt{:}\Sigma x :: I.T\}$, `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.

3.3.4 Built-in operations

The operational semantics of the more 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 $a.\texttt{[}x\texttt{]}$, 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.

3.3.5 Array construction

The array constructor $\texttt{[}t^n \,|\, \texttt{[}f^d\texttt{]]}$ with $\forall i.\, f_i > 0$ and $n = \Pi_{i=1}^{d} f_i$ 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.

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

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 $\texttt{gen}\ x\ \texttt{<}\ t_1\ \texttt{of}\ t_2\ \texttt{with}\ t_3$ 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:

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

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 $\texttt{[|}\,s^d\,\texttt{|}\,\texttt{[}d\texttt{]}\,\texttt{|]}$, 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 $\texttt{[}t_c^p\,\texttt{|}\,\texttt{[}s^d\texttt{]]}$, that in turn will evaluate 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 whole WITH-loop will evaluate to an empty array of shape $t_1 \mathbin{+\!\!+} t_2$ as stated by rule E-GenE. Having no quarks, the empty array has quark type $\bot_Q$ and is thus compatible with any other quark type.

3.3.6 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 $\texttt{loop}\ x_1\ \texttt{<}\ t_1\texttt{,}\ x_2\ \texttt{=}\ t_2\ \texttt{with}\ t_3$, 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:

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

3.3.7 Conditional

Finally, the language provides support for a generalized form of a conditional. Its semantics is shown in Fig. 8. The expression $\texttt{case}\ t\ \texttt{in}\ m$ evaluates to

$$\frac{t \longrightarrow t'}{\texttt{case}\ t\ \texttt{in}\ m \longrightarrow \texttt{case}\ t'\ \texttt{in}\ m}\ (\text{E-Case})$$

$$\texttt{case}\ v\ \texttt{in}\ \texttt{else} \Rightarrow t \longrightarrow t\ (\text{E-Else})$$

$$\frac{r \longrightarrow r'}{\texttt{case}\ v\ \texttt{in}\ r \Rightarrow t\ \texttt{|}\ m \longrightarrow \texttt{case}\ v\ \texttt{in}\ r' \Rightarrow t\ \texttt{|}\ m}\ (\text{E-Range})$$

$$\frac{M(v, rv)}{\texttt{case}\ v\ \texttt{in}\ rv \Rightarrow t\ \texttt{|}\ m \longrightarrow t}\ (\text{E-Match})$$

$$\frac{\neg M(v, rv)}{\texttt{case}\ v\ \texttt{in}\ rv \Rightarrow t\ \texttt{|}\ m \longrightarrow \texttt{case}\ v\ \texttt{in}\ m}\ (\text{E-Next})$$

Fig. 8. Semantics for conditional expressions

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 branches of the form $r \Rightarrow t_b\ \texttt{|}\ m_n$. If the value of t lies in the range r, the conditional evaluates to t_b. Otherwise, the next branch in m_n is tried. In case there is no matching branch, the terminal $\texttt{else} \Rightarrow t_e$ 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 $a.\texttt{[}x\texttt{]}$:

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

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 $\texttt{[}Q\,\texttt{|}\,\texttt{[]}\texttt{]}$. Similarly, we abbreviate a scalar array value $\texttt{[|}\,q\,\texttt{|}\,\texttt{[]}\,\texttt{|]}$ 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 should be straightforward.

Using the WITH-loop, shape-generic algorithms may be specified. 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 × 2 matrices:

```
map : (int → int) → [int|[2,2]] → [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 : Πs :: natvec(2).
        (int → int) → [int|s] → [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 : Πr :: nat. Πs :: natvec(r).
        (int → int) → [int|s] → [int|s]
map 'r 's f a = gen x < shape a of [||[0]|] with f a.[x]
```

To provide an example that uses 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 `cpxmul` 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]]
```

An example for a *non-uniform array operation* is the generalized selection `gsel`. It overcomes the restriction that the length of a selection vector must match the rank of the array selected into. Given a shorter selection vector x and an array a, it selects an array slice of 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 $\vec{0}$ 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 Σ-type indicates precisely that the values of the vectors range between $\vec{0}$ 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 `msel`. It takes an array a and another array i of (legal) selection vectors into a. `msel` then performs a selection into a for every vector in i and yields the array of all results.

```
msel : Πr :: nat. Πs :: natvec(r).
       Πt :: nat. Πu :: natvec(t).
       [int|s] → [Σy :: {natvec(r) in ..s}. numvec(y)|u] →
       [int|u]
msel '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 functions `fold`. 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.

```
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).
         Πl :: {nat in ..r + 1}. Πv :: {natvec(l) in ..drop(r - l,s)}.
         [int|s] → numvec(v) → [int|take(r - l,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 [| | [] |] 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 in the same style as 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 and slight modification of the code.

```
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 of 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 won't evaluate to a value, we now present a type system for static verification 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. Although this style makes the rules short and clear, 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 Γ. It includes type declarations $x : T$, sort declarations $x :: I$, and

$$\Gamma \vdash \texttt{idx} :: *_I \quad (\text{WFS-I}\textsc{dx})$$

$$\frac{\Gamma \vdash i :: \{\texttt{idx in } 0\texttt{..}\}}{\Gamma \vdash \texttt{idxvec(}i\texttt{)} :: *_I} \ (\text{WFS-V}\textsc{ec})$$

$$\frac{\Gamma \vdash I :: *_I \qquad \Gamma \vdash ir :: I_r \qquad \Gamma,\, x :: I \vdash x :: I_r}{\Gamma \vdash \{I \texttt{ in } ir\} :: *_I} \ (\text{WFS-S}\textsc{ubset})$$

Fig. 9. Well-formedness of sorts

additional constraints for confining index terms to specific index ranges, e.g. $x + 1$ `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.

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

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 $\Gamma \models i \texttt{ in } ir$ and $\Gamma \overset{\rightharpoonup}{\models} i \texttt{ in } ir$ 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 the interpretation of the sorts `idx` and `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:

$$r :: \{\texttt{idx in } 0\texttt{..}\},\ s :: \{\texttt{idxvec(}r\texttt{) in vec(}r\texttt{,}1\texttt{)..}\} \overset{\rightharpoonup}{\models} s \texttt{ in vec(}r\texttt{,}0\texttt{)..}$$

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 $\Gamma \vdash i :: I$, WFS-Vec ensures that, for every vector sort `idxvec(`i`)`, i is a non-negative integer. WFS-Subsort accepts only those subset sorts $\{I \texttt{ in } ir\}$ whose bounds in ir have a sort compatible with the base sort I, i.e. they have a common root sort I_r.

$$\frac{\Gamma \vdash i :: \{I \ \texttt{in}\ ir\}}{\Gamma \vdash i :: I} \ (\text{S-Superset})$$

$$\frac{\Gamma \vdash i :: \texttt{idx} \qquad \Gamma \vdash i :: I \qquad \Gamma \models i \ \texttt{in}\ ir}{\Gamma \vdash i :: \{I \ \texttt{in}\ ir\}} \ (\text{S-SSubset})$$

$$\frac{\Gamma \vdash i :: \texttt{idxvec(}i_l\texttt{)} \qquad \Gamma \vdash i :: I \qquad \Gamma \overset{\rightarrow}{\models} i \ \texttt{in}\ ir}{\Gamma \vdash i :: \{I \ \texttt{in}\ ir\}} \ (\text{S-VSubset})$$

$$\frac{\Gamma \vdash i :: \texttt{idxvec(}i_1\texttt{)} \qquad \Gamma \vdash i_2 :: \{\texttt{idx in}\ i_1\}}{\Gamma \vdash i :: \texttt{idxvec(}i_2\texttt{)}} \ (\text{S-VLen})$$

$$\frac{x :: I \in \Gamma}{\Gamma \vdash x :: I} \ (\text{S-Ctx})$$

$$\Gamma \vdash c :: \texttt{idx} \ (\text{S-Idx})$$

$$\frac{\forall j. \Gamma \vdash i_j :: \texttt{idx}}{\Gamma \vdash \texttt{[}i_1, .., i_n\texttt{]} :: \texttt{idxvec(}n\texttt{)}} \ (\text{S-Vect})$$

$$\frac{\Gamma \vdash i_1 :: \{\texttt{idx in}\ 0\texttt{..}\} \qquad \Gamma \vdash i_2 :: \texttt{idx}}{\Gamma \vdash \texttt{vec(}i_1\texttt{,}i_2\texttt{)} :: \texttt{idxvec(}i_1\texttt{)}} \ (\text{S-Vec})$$

$$\frac{\Gamma \vdash i_1 :: \texttt{idxvec(}m\texttt{)} \qquad \Gamma \vdash i_2 :: \texttt{idxvec(}n\texttt{)}}{\Gamma \vdash i_1 \mathbin{+\!\!+} i_2 :: \texttt{idxvec(}m+n\texttt{)}} \ (\text{S-Cat})$$

$$\frac{\Gamma \vdash i_1 :: \{\texttt{idx in}\ 0\texttt{..}n+1\} \qquad \Gamma \vdash i_2 :: \texttt{idxvec(}n\texttt{)}}{\Gamma \vdash \texttt{take(}i_1\texttt{,}i_2\texttt{)} :: \texttt{idxvec(}i_1\texttt{)}} \ (\text{S-Take})$$

$$\frac{\Gamma \vdash i_1 :: \{\texttt{idx in}\ 0\texttt{..}n+1\} \qquad \Gamma \vdash i_2 :: \texttt{idxvec(}n\texttt{)}}{\Gamma \vdash \texttt{drop(}i_1\texttt{,}i_2\texttt{)} :: \texttt{idxvec(}n-i_1\texttt{)}} \ (\text{S-Drop})$$

$$\frac{\Gamma \vdash i_1 :: \texttt{idx} \qquad \Gamma \vdash i_2 :: \texttt{idx}}{\Gamma \vdash f_2\texttt{(}i_1\texttt{,}i_2\texttt{)} :: \texttt{idx}} \ (\text{S-SBin})$$

$$\frac{\Gamma \vdash i_1 :: \texttt{idxvec(}i\texttt{)} \qquad \Gamma \vdash i_2 :: \texttt{idxvec(}i\texttt{)}}{\Gamma \vdash f_2\texttt{(}i_1\texttt{,}i_2\texttt{)} :: \texttt{idxvec(}i\texttt{)}} \ (\text{S-VBin})$$

$$\frac{\Gamma \vdash i :: \texttt{idx}}{\Gamma \vdash i\texttt{..} :: \texttt{idx}} \ (\text{RS-SFrom}) \qquad \frac{\Gamma \vdash i :: \texttt{idx}}{\Gamma \vdash \texttt{..}i :: \texttt{idx}} \ (\text{RS-STo})$$

$$\frac{\Gamma \vdash i_1 :: \texttt{idx} \qquad \Gamma \vdash i_2 :: \texttt{idx}}{\Gamma \vdash i_1\texttt{..}i_2 :: \texttt{idx}} \ (\text{RS-SFromTo})$$

$$\frac{\Gamma \vdash i :: \texttt{idxvec(}i_l\texttt{)}}{\Gamma \vdash i\texttt{..} :: \texttt{idxvec(}i_l\texttt{)}} \ (\text{RS-VFrom}) \qquad \frac{\Gamma \vdash i :: \texttt{idxvec(}i_l\texttt{)}}{\Gamma \vdash \texttt{..}i :: \texttt{idxvec(}i_l\texttt{)}} \ (\text{RS-VTo})$$

$$\frac{\Gamma \vdash i_1 :: \texttt{idxvec(}i_l\texttt{)} \qquad \Gamma \vdash i_2 :: \texttt{idxvec(}i_l\texttt{)}}{\Gamma \vdash i_1\texttt{..}i_2 :: \texttt{idxvec(}i_l\texttt{)}} \ (\text{RS-VFromTo})$$

Fig. 10. The sorting relation

$$\Gamma \vdash \texttt{int} : *_Q \quad \text{(QWF-INT)}$$

$$\frac{\Gamma \vdash T_1 : * \qquad \Gamma \vdash T_2 : *}{\Gamma \vdash T_1 \to T_2 : *_Q} \ \text{(QWF-FUN)}$$

$$\frac{\Gamma \vdash I :: *_I \qquad \Gamma,\, x :: I \vdash T : *}{\Gamma \vdash \Pi x :: I.\, T : *_Q} \ \text{(QWF-PI)}$$

$$\frac{\forall j.\ \Gamma \vdash T_j : *}{\Gamma \vdash \{T^n\} : *_Q} \ \text{(QWF-TUP)}$$

$$\frac{\Gamma \vdash I :: *_I \qquad \Gamma,\, x :: I \vdash T : *}{\Gamma \vdash \Sigma x :: I.\, T : *_Q} \ \text{(QWF-SIGMA)}$$

$$\frac{\Gamma \vdash Q : *_Q \qquad \Gamma \vdash i :: \{\texttt{idxvec(}n\texttt{) in vec(}n\texttt{,}0\texttt{)..}\}}{\Gamma \vdash \texttt{[}Q\texttt{|}i\texttt{]} : *} \ \text{(WF-ARRAY)}$$

$$\frac{\Gamma \vdash i :: \texttt{idx}}{\Gamma \vdash \texttt{num(}i\texttt{)} : *} \ \text{(WF-NUM)}$$

$$\frac{\Gamma \vdash i :: \texttt{idxvec(}n\texttt{)}}{\Gamma \vdash \texttt{numvec(}i\texttt{)} : *} \ \text{(WF-NUMVEC)}$$

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

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 `idx`. But it is also a natural number $\{\texttt{idx in } 0\texttt{..}\}$, a number between 0 and 10 $\{\texttt{idx in } 0\texttt{..}10\}$, and an integer equal to 2 $\{\texttt{idx in } 2\}$.

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 $\{I \texttt{ in } ir\}$ 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 $\{I \texttt{ in } ir\}$. 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 $\texttt{idxvec(}1+2\texttt{)}$ also has sort $\texttt{idxvec(}3\texttt{)}$.

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 requiring only the necessary preconditions. The last 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 `num(`i`)` requires a scalar index term i, whereas singleton vector types `numvec(`i`)` need an index vector. Note that $\bot_Q$ 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.

5.6 Subtyping

The subtype relations on types $\Gamma \vdash T <: T$ and quark types $\Gamma \vdash Q <:_Q Q$, shown in Fig. 12, follow the same mutually recursive pattern. Both relations are reflexive and transitive. The bottom quark type $\bot_Q$ 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 a 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 [?]: An array of type `[`Q_1`|`i`]` may be known in a different context as a `[`Q_2`|`i`]`, with $\Gamma \vdash Q_1 <:_Q Q_2$. 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.

$$\Gamma \vdash Q <:_Q Q \; (\textsc{QSub-Refl})$$

$$\frac{\Gamma \vdash Q_1 <:_Q Q_2 \qquad \Gamma \vdash Q_2 <:_Q Q_3}{\Gamma \vdash Q_1 <:_Q Q_3} \; (\textsc{QSub-Trans})$$

$$\Gamma \vdash \bot_Q <:_Q Q \; (\textsc{QSub-Bot})$$

$$\frac{\Gamma \vdash S_1 <: T_1 \qquad \Gamma \vdash T_2 <: S_2}{\Gamma \vdash T_1 \to T_2 <:_Q S_1 \to S_2} \; (\textsc{QSub-Fun})$$

$$\frac{\Gamma,\, x :: I_2 \vdash x :: I_1 \qquad \Gamma,\, x_2 :: I_2 \vdash T_1[x_1 \mapsto_i x_2] <: T_2}{\Gamma \vdash \Pi x_1 :: I_1.\, T_1 <:_Q \Pi x_2 :: I_2.\, T_2} \; (\textsc{QSub-Pi})$$

$$\frac{\forall j.\, \Gamma \vdash T_j <: S_j}{\Gamma \vdash \{T^n\} <:_Q \{S^n\}} \; (\textsc{QSub-Tup})$$

$$\frac{\Gamma,\, x :: I_1 \vdash x :: I_2 \qquad \Gamma,\, x_1 :: I_1 \vdash T_1 <: T_2[x_2 \mapsto_i x_1]}{\Gamma \vdash \Sigma x_1 :: I_1.\, T_1 <:_Q \Sigma x_2 :: I_2.\, T_2} \; (\textsc{QSub-Sigma})$$

$$\Gamma \vdash T <: T \; (\textsc{Sub-Refl})$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \; (\textsc{Sub-Trans})$$

$$\frac{\Gamma \vdash Q_1 <:_Q Q_2}{\Gamma \vdash \texttt{[}Q_1\,\texttt{|}\,i\texttt{]} <: \texttt{[}Q_2\,\texttt{|}\,i\texttt{]}} \; (\textsc{Sub-ArrQ})$$

$$\frac{\Gamma \vdash i_1 :: \texttt{idxvec(}i\texttt{)} \qquad \Gamma \vdash i_2 :: \{\texttt{idxvec(}i\texttt{) in } i_1\}}{\Gamma \vdash \texttt{[}Q\,\texttt{|}\,i_1\texttt{]} <: \texttt{[}Q\,\texttt{|}\,i_2\texttt{]}} \; (\textsc{Sub-ArrShp})$$

$$\frac{\Gamma \vdash i_1 :: I \qquad \Gamma \vdash i_2 :: \{I \texttt{ in } i_1\}}{\Gamma \vdash S\texttt{(}i_1\texttt{)} <: S\texttt{(}i_2\texttt{)}} \; (\textsc{Sub-Single})$$

$$\Gamma \vdash \texttt{num(}i\texttt{)} <: \texttt{[int|[]]} \; (\textsc{Sub-Num})$$

$$\frac{\Gamma \vdash i :: \texttt{idxvec(}i_l\texttt{)}}{\Gamma \vdash \texttt{numvec(}i\texttt{)} <: \texttt{[int|[}l\texttt{]]}} \; (\textsc{Sub-Numvec})$$

Fig. 12. Subtyping on types and quark types

5.7 Type checking

Now that we treated all the prerequisites, we can define the typing relation $\Gamma \vdash t : T$ and the quark typing relation $\Gamma \vdash q :_Q Q$. 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.

According to rule T-Val, type checking of non-empty array values $\texttt{[|}\, q^p \,\texttt{|}\, \texttt{[}s^d\texttt{]}\, \texttt{|]}$ requires to verify that each quark q_i has the same quark type Q. For arrays of abstractions, Q has the form $T_1 \to T_2$. 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

$$\frac{\Gamma \vdash q :_Q Q_1 \qquad \Gamma \vdash Q_1 <:_Q Q_2}{\Gamma \vdash q :_Q Q_2}\ (\textsc{QT-Sub})$$

$$\Gamma \vdash c :_Q \mathtt{int}\ (\textsc{QT-Int})$$

$$\frac{\Gamma,\ x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 .\, t :_Q T_1 \to T_2}\ (\textsc{QT-Abs})$$

$$\frac{\Gamma,\ x :: I \vdash t : T}{\Gamma \vdash \lambda' x :: I .\, t :_Q \Pi x :: I .\, T}\ (\textsc{QT-Pi})$$

$$\frac{\forall j.\ \Gamma \vdash v_j : T_j}{\Gamma \vdash \{v^n\} :_Q \{T^n\}}\ (\textsc{QT-Tup})$$

$$\frac{\Gamma \vdash \Sigma x :: I .\, T : *_Q \qquad \Gamma \vdash i :: I \qquad \Gamma \vdash t : T[x \mapsto_i i]}{\Gamma \vdash \{' i , t : \Sigma x :: I .\, T\} :_Q \Sigma x :: I .\, T}\ (\textsc{QT-Sigma})$$

$$\frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash t : T_2}\ (\textsc{T-Sub})$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (\textsc{T-Ctx})$$

$$\frac{n > 0 \qquad \forall j.\ \Gamma \vdash q_j :_Q Q}{\Gamma \vdash \mathtt{[|}\, q^n \,|\, \mathtt{[}s^d\mathtt{]} \,\mathtt{|]} : \mathtt{[}Q \,|\, \mathtt{[}s^d\mathtt{]}\mathtt{]}}\ (\textsc{T-Val})$$

$$\Gamma \vdash \mathtt{[|} \,|\, \mathtt{[}s^d\mathtt{]} \,\mathtt{|]} : \mathtt{[}\bot_Q \,|\, \mathtt{[}s^d\mathtt{]}\mathtt{]}\ (\textsc{T-ValE})$$

$$\Gamma \vdash \mathtt{[|}\, c \,|\, \mathtt{[]} \,\mathtt{|]} : \mathtt{num(}c\mathtt{)}\ (\textsc{T-Num})$$

$$\Gamma \vdash \mathtt{[|}\, c^n \,|\, \mathtt{[}n\mathtt{]} \,\mathtt{|]} : \mathtt{numvec([}c^n\mathtt{])}\ (\textsc{T-Numvec})$$

$$\frac{\Gamma \vdash t_1 : \mathtt{[}T_1 \to T_2 \,|\, \mathtt{[]]} \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}\ (\textsc{T-App})$$

$$\frac{\Gamma \vdash t : \mathtt{[}\Pi x :: I .\, T \,|\, \mathtt{[]]} \qquad \Gamma \vdash i :: I}{\Gamma \vdash t\ ' i : T[x \mapsto_i i]}\ (\textsc{T-IApp})$$

$$\frac{\forall j.\ \Gamma \vdash t_j : T_j}{\Gamma \vdash \{t^n\} : \mathtt{[}\{T^n\} \,|\, \mathtt{[]]}}\ (\textsc{T-Tup})$$

$$\frac{\Gamma \vdash \Sigma x :: I .\, T : *_Q \qquad \Gamma \vdash i :: I \qquad \Gamma \vdash t : T[x \mapsto_i i]}{\Gamma \vdash \{' i , t : \Sigma x :: I .\, T\} : \mathtt{[}\Sigma x :: I .\, T \,|\, \mathtt{[]]}}\ (\textsc{T-ITup})$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma,\ x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let}\ x \mathtt{=} t_1\ \mathtt{in}\ t_2 : T_2}\ (\textsc{T-Let})$$

$$\frac{\Gamma \vdash t_1 : \mathtt{[}\{T^n\} \,|\, \mathtt{[]]} \qquad \Gamma,\ x_1 : T_1, .., x_n : T_n \vdash t_2 : T_{n+1}}{\Gamma \vdash \mathtt{let}\ \{x^n\} \mathtt{=} t_1\ \mathtt{in}\ t_2 : T_{n+1}}\ (\textsc{T-Unpack})$$

$$\frac{\Gamma \vdash t_1 : \mathtt{[}\Sigma x :: I .\, T \,|\, \mathtt{[]]} \qquad \Gamma,\ x_i :: I,\ x : T[x \mapsto_i x_i] \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let}\ \{x_i , x\} \mathtt{=} t_1\ \mathtt{in}\ t_2 : T_2}\ (\textsc{T-IUnpack})$$

Fig. 13. Basic typing rules

functions works analogously. A dependent pair $\{'i\texttt{,}t\texttt{:}\Sigma x :: I.T\}$ 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 $\bot_Q$, 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 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 $\texttt{[}Q \mid f +\!\!+ i_c\texttt{]}$. In the special case where all cells of a vector are singleton scalars, rule T-ArrNumvec gives the array the appropriate singleton vector type. Typing of a WITH-loop `gen` x `<` t_1 `of` t_2 `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 $\texttt{[}Q \mid i_2\texttt{]}$, where i_2 is also the value of the cell shape t_2, then the WITH-loop has type $\texttt{[}Q \mid i_1 +\!\!+ i_2\texttt{]}$.

$$\frac{\Gamma \vdash t : \texttt{[}Q\,\texttt{|}\,i\texttt{]} \qquad \Gamma \vdash i :: \texttt{idxvec(}i_l\texttt{)}}{\Gamma \vdash \texttt{rank}\ t : \texttt{num(}i_l\texttt{)}}\ (\textsc{T-Rank})$$

$$\frac{\Gamma \vdash t : \texttt{[}Q\,\texttt{|}\,i\texttt{]}}{\Gamma \vdash \texttt{shape}\ t : \texttt{numvec(}i\texttt{)}}\ (\textsc{T-Shape})$$

$$\frac{\Gamma \vdash t : \texttt{numvec(}i\texttt{)} \qquad \Gamma \vdash i :: \texttt{idxvec(}i_l\texttt{)}}{\Gamma \vdash \texttt{length}\ t : \texttt{num(}i_l\texttt{)}}\ (\textsc{T-Length})$$

$$\frac{\Gamma \vdash t : \texttt{[}\{\texttt{num(}i_1\texttt{)},\texttt{num(}i_2\texttt{)}\}\,\texttt{|}\,\texttt{[]]}}{\Gamma \vdash f_2\ t : \texttt{num(}f_2\texttt{(}i_1\texttt{,}i_2\texttt{))}}\ (\textsc{T-BinS})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \texttt{[}\{\texttt{numvec(}i_1\texttt{)},\texttt{numvec(}i_2\texttt{)}\}\,\texttt{|}\,\texttt{[]]}\\ \Gamma \vdash i_1 :: \texttt{idxvec(}i\texttt{)} \qquad \Gamma \vdash i_2 :: \texttt{idxvec(}i\texttt{)}\end{array}}{\Gamma \vdash f_2\ t : \texttt{numvec(}f_2\texttt{(}i_1\texttt{,}i_2\texttt{))}}\ (\textsc{T-BinV})$$

$$\frac{\Gamma \vdash t : \texttt{[}\{\texttt{[int|}i\texttt{]},\texttt{[int|}i\texttt{]}\}\,\texttt{|}\,\texttt{[]]}}{\Gamma \vdash f_2\ t : \texttt{[int|}i\texttt{]}}\ (\textsc{T-Bin})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \texttt{[}\{\texttt{[}Q\,\texttt{|}\,i_s\texttt{]},\texttt{numvec(}i\texttt{)}\}\,\texttt{|}\,\texttt{[]]} \qquad \Gamma \vdash i_s :: \texttt{idxvec(}i_l\texttt{)}\\ \Gamma \vdash i :: \{\texttt{idxvec(}i_l\texttt{)}\ \texttt{in}\ \texttt{vec(}i_l\texttt{,}0\texttt{)..}i_s\}\end{array}}{\Gamma \vdash \texttt{sel}\ t : \texttt{[}Q\,\texttt{|}\,\texttt{[]]}}\ (\textsc{T-Sel})$$

$$\frac{\Gamma \vdash t : \texttt{[}\{\texttt{num(}i_l\texttt{)},\texttt{num(}i\texttt{)}\}\,\texttt{|}\,\texttt{[]]} \qquad \Gamma \vdash i_l :: \{\texttt{idx}\ \texttt{in}\ 0\texttt{..}\}}{\Gamma \vdash \texttt{vec}\ t : \texttt{numvec(vec(}i_l\texttt{,}i\texttt{))}}\ (\textsc{T-Vec})$$

$$\frac{\Gamma \vdash t : \texttt{[}\{\texttt{numvec(}i_1\texttt{)},\texttt{numvec(}i_2\texttt{)}\}\,\texttt{|}\,\texttt{[]]}}{\Gamma \vdash \texttt{++}\ t : \texttt{numvec(}i_1 \mathbin{+\!\!+} i_2\texttt{)}}\ (\textsc{T-Cat})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \texttt{[}\{\texttt{num(}i\texttt{)},\texttt{numvec(}i_v\texttt{)}\}\,\texttt{|}\,\texttt{[]]}\\ \Gamma \vdash i_v :: \texttt{idxvec(}i_l\texttt{)} \qquad \Gamma \vdash i :: \{\texttt{idx}\ \texttt{in}\ 0\texttt{..}i_l + 1\}\end{array}}{\Gamma \vdash \texttt{take}\ t : \texttt{numvec(take(}i\texttt{,}i_v\texttt{))}}\ (\textsc{T-Take})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \texttt{[}\{\texttt{num(}i\texttt{)},\texttt{numvec(}i_v\texttt{)}\}\,\texttt{|}\,\texttt{[]]}\\ \Gamma \vdash i_v :: \texttt{idxvec(}i_l\texttt{)} \qquad \Gamma \vdash i :: \{\texttt{idx}\ \texttt{in}\ 0\texttt{..}i_l + 1\}\end{array}}{\Gamma \vdash \texttt{drop}\ t : \texttt{numvec(drop(}i\texttt{,}i_v\texttt{))}}\ (\textsc{T-Drop})$$

$$\frac{\forall j.\ \Gamma \vdash t_j : \texttt{[}Q\,\texttt{|}\,i\texttt{]}}{\Gamma \vdash \texttt{[}t^p\,\texttt{|}\,\texttt{[}c^n\texttt{]]} : \texttt{[}Q\,\texttt{|}\,\texttt{[}c^n\texttt{]} \mathbin{+\!\!+} i\texttt{]}}\ (\textsc{T-Arr})$$

$$\frac{\forall j.\ \Gamma \vdash t_j : \texttt{num(}i_j\texttt{)}}{\Gamma \vdash \texttt{[}t^n\,\texttt{|}\,\texttt{[}n\texttt{]]} : \texttt{numvec([}i^n\texttt{])}}\ (\textsc{T-ArrNumvec})$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \texttt{numvec(}i_1\texttt{)} \qquad \Gamma \vdash i_1 :: \{\texttt{idxvec(}n\texttt{)}\ \texttt{in}\ \texttt{vec(}n\texttt{,}0\texttt{)..}\}\\ \Gamma \vdash t_2 : \texttt{numvec(}i_2\texttt{)} \qquad \Gamma \vdash i_2 :: \{\texttt{idxvec(}m\texttt{)}\ \texttt{in}\ \texttt{vec(}m\texttt{,}0\texttt{)..}\}\\ \Gamma,\ x :: \{\texttt{idxvec(}n\texttt{)}\ \texttt{in}\ \texttt{vec(}n\texttt{,}0\texttt{)..}i_1\},\ x : \texttt{numvec(}x\texttt{)} \vdash t_3 : \texttt{[}Q\,\texttt{|}\,i_2\texttt{]}\end{array}}{\Gamma \vdash \texttt{gen}\ x\ \texttt{<}\ t_1\ \texttt{of}\ t_2\ \texttt{with}\ t_3 : \texttt{[}Q\,\texttt{|}\,i_1 \mathbin{+\!\!+} i_2\texttt{]}}\ (\textsc{T-Gen})$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \texttt{numvec(}i\texttt{)} \qquad \Gamma \vdash i :: \{\texttt{idxvec(}n\texttt{)}\ \texttt{in}\ \texttt{vec(}n\texttt{,}0\texttt{)..}\} \qquad \Gamma \vdash t_2 : T\\ \Gamma,\ x_1 :: \{\texttt{idxvec(}n\texttt{)}\ \texttt{in}\ \texttt{vec(}n\texttt{,}0\texttt{)..}i\},\ x_1 : \texttt{numvec(}x_1\texttt{)}, x_2 : T \vdash t_3 : T\end{array}}{\Gamma \vdash \texttt{loop}\ x_1\ \texttt{<}\ t_1\texttt{,}\ x_2\ \texttt{=}\ t_2\ \texttt{with}\ t_3 : T}\ (\textsc{T-Loop})$$

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

$$\frac{\Gamma \vdash t : S(i) \qquad \Gamma \,|\, S(i) \vdash m : T_m}{\Gamma \vdash \texttt{case}\ t\ \texttt{in}\ m : T_m}\ (\text{T-}\textsc{Case})$$

$$\frac{\Gamma \vdash t : T}{\Gamma \,|\, S(i) \vdash \texttt{else} \Rightarrow t : T}\ (\text{T-}\textsc{Else})$$

$$\frac{\Gamma \,|\, S(i) \vdash r ::_r ir \qquad \Gamma, i\ \texttt{in}\ ir \vdash t : T \qquad \Gamma \,|\, S(i) \vdash m : T}{\Gamma \,|\, S(i) \vdash r \Rightarrow t\ \texttt{|}\ m : T}\ (\text{T-}\textsc{Range})$$

$$\frac{\Gamma \vdash t : S(i_r) \qquad \Gamma \vdash i_r :: I \qquad \Gamma \vdash i :: I}{\Gamma \,|\, S(i) \vdash t ::_r i_r}\ (\text{IR-}\textsc{Eq})$$

$$\frac{\Gamma \vdash t : S(i_r) \qquad \Gamma \vdash i_r :: I \qquad \Gamma \vdash i :: I}{\Gamma \,|\, S(i) \vdash t\texttt{..} ::_r i_r\texttt{..}}\ (\text{IR-}\textsc{From})$$

$$\frac{\Gamma \vdash t : S(i_r) \qquad \Gamma \vdash i_r :: I \qquad \Gamma \vdash i :: I}{\Gamma \,|\, S(i) \vdash \texttt{..}t ::_r \texttt{..}i_r}\ (\text{IR-}\textsc{To})$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : S(i_1) \qquad \Gamma \vdash t_2 : S(i_2) \\ \Gamma \vdash i_1 :: I \qquad \Gamma \vdash i_2 :: I \qquad \Gamma \vdash i :: I\end{array}}{\Gamma \,|\, S(i) \vdash t_1\texttt{..}t_2 ::_r i_1\texttt{..}i_2}\ (\text{IR-}\textsc{FromTo})$$

Fig. 15. Typing rules for conditional expressions

Similarly, typing of a loop `loop` x_1 `<` t_1`,` x_2 `=` t_2 `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 `case` t `in` m are typed according to the typing 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 to the branching condition, i.e. that all ranges are are integer singletons of the same shape as t. For this purpose, the auxiliary typing relation $\Gamma \,|\, S(i) \vdash m : T$ 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. Since the branch is only evaluated if the value of the branching condition lies within the range r, it checks the branch with the additional property i `in` ir. The branch must then have the same type as the other branches. The type of the terminal branch `else` $\Rightarrow t_e$ is just the type of t_e.

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 value or $\exists t'.\, t \longrightarrow t'$.*

Proof: By induction on typing derivations (see proof in appendix of extended technical report [**?**]).

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 [**?**]).

We have specified the typing rules in a declarative style, which is concise but does not allow for a 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 rules 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* $\Gamma \models i \texttt{ in } ir$ and $\Gamma \overset{\rightarrow}{\models} i \texttt{ in } ir$. They provide proof that under a given set of assumptions Γ 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 Γ 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 don't contain sort declarations of subset sorts. These are transformed into a declaration of the root sort and a subsequent sequence of constraints, e.g. $x :: \{\texttt{idx in } 0\texttt{..}\} \leadsto x :: \texttt{idx}, x \texttt{ in } 0\texttt{..}$. The type declarations in Γ 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.

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

Scalar judgments $\Gamma \models i \texttt{ in } ir$ 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 [?,?] can then refute the negated property, thereby validating the judgment.

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

The decision procedure for vector judgments $\Gamma \overset{\rightarrow}{\models} i \texttt{ in } ir$ takes both sets $\mathcal{S}(\Gamma)$ and $\mathcal{V}(\Gamma)$ into account. Similar to the approach for scalars, we rewrite the problem such that is 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 [?] 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* φ_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 whose 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.

$$\begin{array}{ll}
\mathcal{T}\,[\![x]\!][i] & = x[i] \\
\mathcal{T}\,[\![s^t]\!][i] & = s \\
\mathcal{T}\,[\![f_2(v_1, v_2)]\!][i] & = f_2(\mathcal{T}\,[\![v_1]\!][i], \mathcal{T}\,[\![v_2]\!][i])
\end{array}$$

$$\begin{array}{ll}
\mathcal{P}\,[\![i_1 \texttt{ in } i_2]\!] & = (\forall j.\, 0 \leq j < |i_1| \Rightarrow \mathcal{T}\,[\![i_1]\!][j] = \mathcal{T}\,[\![i_2]\!][j]) \\
\mathcal{P}\,[\![i_1 \texttt{ in } i_2\texttt{..}]\!] & = (\forall j.\, 0 \leq j < |i_1| \Rightarrow \mathcal{T}\,[\![i_2]\!][j] \leq \mathcal{T}\,[\![i_1]\!][j]) \\
\mathcal{P}\,[\![i_1 \texttt{ in } \texttt{..}i_2]\!] & = (\forall j.\, 0 \leq j < |i_1| \Rightarrow \mathcal{T}\,[\![i_1]\!][j] < \mathcal{T}\,[\![i_2]\!][j]) \\
\mathcal{P}\,[\![i_1 \texttt{ in } i_2\texttt{..}i_3]\!] & = (\forall j.\, 0 \leq j < |i_1| \Rightarrow \mathcal{T}\,[\![i_2]\!][j] \leq \mathcal{T}\,[\![i_1]\!][j] \land \mathcal{T}\,[\![i_1]\!][j] < \mathcal{T}\,[\![i_3]\!][j])
\end{array}$$

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 [?], the quantifiers can be correctly eliminated from this formula by first converting into negated normal form and subsequently instantiating the quantifiers.

$$\begin{array}{l}
d :: \texttt{idx},\ d \texttt{ in } 0\texttt{..},\ s :: \texttt{idxvec(}d\texttt{)},\ s \texttt{ in vec(}d\texttt{,}1\texttt{)..} \overset{\rightarrow}{\models} s \texttt{ in vec(}d\texttt{,}0\texttt{)..} \\
\Leftrightarrow d \geq 0 \land (\forall i.\, 0 \leq i < d \Rightarrow s[i] \geq 1) \land \neg(\forall i.\, 0 \leq i < d \Rightarrow 0 \leq s[i])
\end{array}$$

In general, a vector judgment $\Gamma \overset{\rightarrow}{\models} i \texttt{ in } ir$ 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. E.g. for vectors $x :: \texttt{idxvec(}n\texttt{)}, y :: \texttt{idxvec(}n+5\texttt{)}$ the property $x \texttt{ in drop(}5\texttt{,}y\texttt{)}$ would translate to $(\forall i.\, 0 \leq i < n \Rightarrow x[i] = y[i+5])$. Unfortunately, it was shown in [?] that extending the array property fragment with arithmetic expressions over universally quantified index variables gives 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 can't 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`.

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

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

$$\begin{array}{l}
r :: \texttt{idx},\ r \texttt{ in } 0\texttt{..},\ l :: \texttt{idx},\ l \texttt{ in } 0\texttt{..}r+1, \\
s :: \texttt{idxvec(}r\texttt{)},\ s \texttt{ in vec(}r\texttt{,}0\texttt{)..},\ v :: \texttt{idxvec(}l\texttt{)},\ v \texttt{ in vec(}l\texttt{,}0\texttt{)..take(}l\texttt{,}s\texttt{)}, \\
y :: \texttt{idxvec(}r-l\texttt{)},\ y \texttt{ in vec(}r-l\texttt{,}0\texttt{)..drop(}l\texttt{,}s\texttt{)} \overset{\rightarrow}{\models} v +\!\!+ y \texttt{ in vec(}r\texttt{,}0\texttt{)..}s
\end{array}$$

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 $\texttt{take(}l\texttt{,}s\texttt{)}$ and $\texttt{drop(}l\texttt{,}s\texttt{)}$ are then substituted with s_1 and s_2, respectively. s itself is consistently replaced with $s_1 +\!\!+ s_2$.

$$\begin{array}{l}
r :: \texttt{idx},\ r \texttt{ in } 0\texttt{..},\ l :: \texttt{idx},\ l \texttt{ in } 0\texttt{..}r+1, \\
s_1 :: \texttt{idxvec(}l\texttt{)},\ s_2 :: \texttt{idxvec(}r-l\texttt{)},\ s_1 +\!\!+ s_2 \texttt{ in vec(}r\texttt{,}0\texttt{)..}, \\
v :: \texttt{idxvec(}l\texttt{)},\ v \texttt{ in vec(}l\texttt{,}0\texttt{)..}s_1,\ y :: \texttt{idxvec(}r-l\texttt{)},\ y \texttt{ in vec(}r-l\texttt{,}0\texttt{)..}s_2 \\
\overset{\rightarrow}{\models} v +\!\!+ y \texttt{ in vec(}r\texttt{,}0\texttt{)..}s_1 +\!\!+ s_2
\end{array}$$

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

$$\begin{array}{l}
r :: \texttt{idx},\ r \texttt{ in } 0\texttt{..},\ l :: \texttt{idx},\ l \texttt{ in } 0\texttt{..}r+1, \\
s_1 :: \texttt{idxvec(}l\texttt{)},\ s_2 :: \texttt{idxvec(}r-l\texttt{)},\ s_1 \texttt{ in vec(}l\texttt{,}0\texttt{)..},\ s_2 \texttt{ in vec(}r-l\texttt{,}0\texttt{)..}, \\
v :: \texttt{idxvec(}l\texttt{)},\ v \texttt{ in vec(}l\texttt{,}0\texttt{)..}s_1,\ y :: \texttt{idxvec(}r-l\texttt{)},\ y \texttt{ in vec(}r-l\texttt{,}0\texttt{)..}s_2 \\
\overset{\rightarrow}{\models} v \texttt{ in vec(}l\texttt{,}0\texttt{)..}s_1,\ y \texttt{ in vec(}r-l\texttt{,}0\texttt{)..}s_2
\end{array}$$

Elimination of structural operations fails if the constraints don't 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.

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

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

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

In the property $y +\!\!+\, v$ `in vec(`r`,0)..`$s_1 +\!\!+\, s_2$, the vectors y and s_1 have length $r - l$ and length l, respectively. The scalar constraints don't allow to derive whether $r - l < l$, $r - l = l$, or $r - l > l$ and thus the property can't 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 `gsel` 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 `in` v_2`..`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 as an array property 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 [?], APL [?,?], J [?] or Nial [?] 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 [?], 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 [?,?,?]. 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 [?] and Nesl [?]. 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 [?,?]. 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 [?]. 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 [?] 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 [?,?], 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 [?].

Most closely related to our approach are more light-weight approaches such as Xi and Pfenning's DML [?], Xi's *applied type system* [?], and Zenger's *indexed types* [?]. 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 $\mathbb{R}^{m\times n} \times \mathbb{R}^{n\times p} \to \mathbb{R}^{m\times p}$. 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 [?] 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 [?]. 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, provided we know that the execution of otherwise dead code does not cause a run time error, this code can safely be eliminated even with a call-by-value semantics. Similarly, we may replace selections into arrays defined by means of WITH-loops with the selected element's definition, thereby achieving deforestation. Finally, in combination with a memory management scheme based on run time reference counting or a linear type system, we may often perform destructive array updates even in our context of immutable arrays. The structural information in the dependent types will help the compiler to identify potentially reusable arrays. Eventually, a substantially revised and extended future version of SaC may incorporate the essential concepts of Qube.

Acknowledgments We would like to thank Florian Büther and Markus Weigel for contributing to the compiler for the Qube language. We also thank Johannes Blume for many interesting discussions about resolution of vector constraints.