Dependently Typed Array Programs Don't Go Wrong

Abstract

Kai Trojahner Clemens Grelck

University of Lübeck
Institute of Software Technology and Programming Languages
{trojahner,grelck}@isp.uni-luebeck.de

1. Introduction

The array programming paradigm adopts arrays as the fundamental data structures of computation. Such arrays may be vectors, matrices, tensors, or objects of even higher dimension. In particular, arrays may also be scalar values such as the integers which form the important special case of arrays without any axes. Array operations work on entire arrays instead of just single elements. This makes array programs highly expressive and introduces data-parallelism in a natural way. Hence, array programs lend themselves well for parallel execution on parallel computers such as recent multi-core processors (Sutter 2005; Grelck 2005). Prominent examples of array languages are APL (Iverson 1962), J (Iverson 1995), MATLAB (Moler et al. 1987), and, more recently, SAC (Grelck and Scholz 2006).

A powerful concept found in array programming languages is shape-generic programming: algorithms may be specified for arrays of arbitrary size and even an arbitrary number of axes. For example, array addition works for scalars as well as for vectors and matrices. However, this flexibility introduces some non-trivial constraints between function arguments: *Uniform array operations* such as element-wise addition require both their arguments to have exactly the same number of axes and the same size. The situation is even more complicated for operations like array selection: The length of the selection vector (a vector of indices) must equal the number of array axes and the *values* of said indices must range within the array bounds.

Interpreted array languages like APL, J, and MATLAB come with a large number of built-in operations. Each operation performs dynamic consistency checks on the structural properties of its arguments. In contrast, SAC is a compiled language aimed at providing utmost computing speed. Clearly, in this setting dynamic checks are undesirable since their repeated execution is costly in terms of run-time overhead. Moreover, uncertainty about whether or not program constraints are met hampers program optimization, leading to poor performance characteristics. Finally, run-time checks do not provide any confidence in program correctness since a program may run correctly for a long time before showing any erroneous behavior. For all these reasons, it is desirable to statically verify the various program constraints by means of formal methods.

Type systems are light-weight formal methods for program verification. A program that has been shown to be well-typed at compile-time is guaranteed not to exhibit certain undesired behavior at run-time. The amount of program errors ruled out by a type system depends on both the programming language to be checked and the expressive power of its types. A language is called type-safe if its operational semantics ensures that well-typedness is preserved by the reduction rules and that the evaluation of well-typed programs never gets into an undefined state in which no further reduction rule applies (Pierce 2002).

In this paper, we present a special type system for static verification of array programs. The system uses families of array types that do not only reflect the type of an array's elements but also contain an expression describing its shape. For specific arrays such as shape vectors, singleton types are employed to capture a vector's value in its type. The type system is based on a novel variant of dependent types: Vectors of integers are used to index into the family of array types. These vectors are themselves indexed from a sort family using an integer. Like in other approaches using indexed types such as DML (Xi and Pfenning 1999), type checking then proceeds by checking constraints on the index expressions, which is decidable. However, dealing with index vectors requires machinery beyond classical theorem provers for presburger arithmetic. To illustrate the approach, we introduce the nucleus of a programming language that allows for the convenient specification of higher-order shape-generic functional array programs. We aim at providing full type-safety, i.e. that all well-typed array programs will yield a value. In short: Dependently typed array programs don't go wrong!

2. A Foundation for Type-Safe Array Programming

The appropriate abstraction for treating the various kinds of arrays uniformly are true multidimensional arrays. As shown in Fig. 1, each array is characterized by its *rank* denoting the number of axes, and its *shape vector* describing the extent of each axis. Since the shape vector contains a natural number for each axis, the rank is implicitly encoded. The elements contained in an array are represented as a potentially empty sequence of *quarks* called the *data vector*. Quarks are the standard values of functional programming languages: constants, function values, and tuples (omitted in this abstract), but here they cannot exist without arrays. Together, the shape vector and the data vector form a unique representation of arrays. As shown in Fig. 2, array terms and quarks are defined mutually recursive.

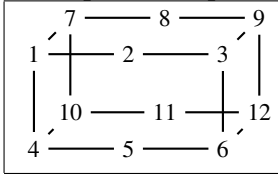
Array	Rank	Shape vector
1	0	[]
1 2 3	1	[3]
1 2 3 4 5 6	2	[2 3]
	3	[2 2 3]

Figure 1. Multidimensional arrays are characterized by their rank and shape vector.

For example, a 2×2 matrix of integers may be represented as $\langle (2,2), (1,2,3,4) : \texttt{int} \rangle$. Further examples are shown in Fig. 3. The type annotation describes the type of the quarks in the data vector. This is necessary to determine the type of the entire array if it does not contain any quarks. Resembling the relationship between quarks and array terms, there are quark types and array types. The array types of a given quark type form a type family in which we select specific types using a type index vector representing the array shape. The type of our above example would thus be $[\texttt{int}|(2,2)]$. Purely for reasons of readability, we may freely abbreviate the types of scalar arrays by writing Q instead of $[Q|()]$.

Using the abstraction quark $\lambda x.\, e$, we can define arrays of functions such as the identity function on scalar integers. Corresponding to the abstraction quark, there is also a quark type $T \to T$ representing function types. Since all array elements must have the same type, the type of the variable in the abstraction is unnecessary:

$$\langle (), (\lambda x.\, x) : \texttt{int} \to \texttt{int} \rangle : [\texttt{int} \to \texttt{int}|()].$$

Only scalar arrays of functions like the example above may be applied to an argument. We define β-reduction on arrays by syntactical substitution whose definition omitted in this abstract. Note that the following reduction does not take the type of the function argument into account:

$$\langle (), (\lambda x.\, t_1) : A \to B \rangle\ t_2 \to [x \mapsto t_2]t_1.$$

Finally, the index abstraction quark $\lambda' x.\, t$ serves to abstract a type index out of a term. Its type is the dependent function quark type $\Pi x : I.\, T$. The index variables may then be used to form index terms. However, like in other approaches using indexed types like DML (Xi and Pfenning 1999), the grammar of these index expressions is limited to retain decidability. The application of dependent functions $t\ 'i$ substitutes the index term i into both the types and terms of t:

$$\langle (), (\lambda' x.\, t) : \Pi x\!:\!I.\, T \rangle\ 'i \to [x \mapsto i]t.$$

I	::=	$\texttt{nat} \mid \texttt{natvec}(i) \mid \{I\ \texttt{in}\ ..i\}$	Index sorts
i	::=	$x \mid c \mid i \texttt{+} i \mid c^i \mid (c^*) \mid i \vec{+} i$	Index terms
	$\mid$	$i \texttt{++} i \mid \texttt{take}(i,i) \mid \texttt{drop}(i,i)$	
T	::=	$[Q\|i] \mid \texttt{num}(i) \mid \texttt{numvec}(i)$	Array types
Q	::=	$\texttt{int} \mid T \to T \mid \Pi x\!:\!I.\, T$	Quark types
t	::=	$\langle (c^*), (q^*) : Q \rangle \mid t\ t \mid t\ 'i$	Array terms
	$\mid$	$\texttt{gen}\ x \texttt{ < } t\ \texttt{with}\ t \mid t[t]$	
q	::=	$c \mid \lambda x.\, e \mid \lambda' x.\, e$	Quarks
v	::=	$\langle (c^*), (q^*) : Q \rangle$	Array values

Figure 2. A simplified syntax for typed array programs

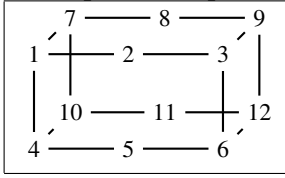
Array	Array expression
1	$\langle (), (1) : \texttt{int} \rangle$
1 2 3	$\langle (3), (1,2,3) : \texttt{int} \rangle$
1 2 3 4 5 6	$\langle (2,3), (1,2,3,4,5,6) : \texttt{int} \rangle$
	$\langle (2,2,3), (1,..,12) : \texttt{int} \rangle$

Figure 3. Arrays represented as array expressions.

Using dependent function types, we may, for the first time in the history of array programming, give an accurate type to the shape-generic addition operation for integer arrays of arbitrary but equal rank and shape:

$$\texttt{add} : \Pi r\!:\!\texttt{nat}.\, \Pi s\!:\!\texttt{natvec}(r).\, [\texttt{int}|s] \to [\texttt{int}|s] \to [\texttt{int}|s].$$

The two layers of index abstraction are typical for shape-generic array programs: we first abstract out the rank r to select the sort of the shape vector s from the sort family of index vectors. Then, the second Π type abstracts out the shape s which is used to index the array types.

The definition of the shape-generic array addition extends an addition operation $+$ for integer scalars to arbitrary arrays using a `gen` expression. To simplify the notation, we use a supercombinator notation that is equivalent to a nesting of scalar function definitions.

$$\texttt{add}\ 'd\ 's\ a\ b = \texttt{gen}\ x \texttt{ < } \texttt{shape}\ a\ \texttt{with}\ a[x] + b[x].$$

The expression $\texttt{gen}\ x \texttt{ < } s\ \texttt{with}\ e$ is a simplified variant of the WITH-loop, the versatile array comprehension used in SAC. It takes a new identifier x, a non-negative vector of integers s, and a scalar-valued expression e that may contain x as a free variable. The expression evaluates to an array of shape s in which the element at index iv is computed by substituting x in e with iv. In the example, we use the built-in function `shape` to determine the shape of the result array based on the shape of the arguments. We may not simply use the index identifier s here because indices will not be evaluated at run-time. The selection operation $a[iv]$ takes an array a and a vector iv whose length equals the rank of a and whose value denotes a valid position within a.

Even for this simple example, verification requires many constraints between array ranks, shapes, and values to be checked:

1. `gen` requires `shape` a to have rank one,
2. `gen` demands the value of `shape` a to be non-negative,
3. $a[x]$ requires x to have rank one,
4. $a[x]$ requires the length of x to match the rank of a,
5. $a[x]$ requires a value of x between $\vec{0}$ and `shape` a,
6. $b[x]$ similar,
7. $+$ demands rank zero arguments,
8. `gen` expects $a[x] + b[x]$ to have rank zero,
9. to meet the specification, the result must have the same shape as a.

The array type $[Q|i]$ contains information about an array's shape, but not about its value. To capture the value of integer arrays, we use two families of singleton types. The members of $\texttt{num}(i)$ and $\texttt{numvec}(i)$ are integer scalars and integer vectors, re-

spectively. For example, $\texttt{numvec}((2,3))$ is the type of the value $\langle(2),(2,3) \texttt{ : int}\rangle$. Every $\texttt{num}(x)$ is also a $[\texttt{int}|()]$ and every $\texttt{numvec}(i)$ is also a $[\texttt{int}|(l)]$ where l denotes the sort index of the index vector i. In this context, `shape` maps array types to singleton types, e.g. in the example `add`, `shape` a gives a $\texttt{numvec}(s)$ because a is of type $[\texttt{int}|s]$.

Singleton types are employed whenever a value must be asserted to lie within a specific range. For example, the selection vector in the selection must be of type `numvec` to assert the selection is in-bounds. Furthermore, in an expression `gen` x `<` s `with` e, s must be a $\texttt{numvec}(s')$ such that the compiler can verify the new array is not meant to have negative extent. During type checking of e, x is then regarded as a $\texttt{numvec}(x')$ where x' is an index vector ranging between $\vec{0}$ and s'.

By merely classifying `shape` a and x as `numvec`, we have already found a proof for the constraints 1 and 3 in the above list. However, in general the constraints must be verified using a theorem prover for formulas in *presburger vector arithmetic*, an extension of presburger arithmetic to vectors of integers having arbitrary length. For example, constraint 2 requires to verify the following formula:

$$\forall d.\, d \geq 0 \Longrightarrow \forall s^d.\, s \geq_\wedge 0^d \Rightarrow s \geq_\wedge 0^d.$$

In this formula, s^d means that s is an arbitrary vector of length d. 0^d creates a vector of zeros of length d. Finally, $\geq_\wedge$ denotes the conjunctive extension of the relation symbol $\geq$, meaning $a \geq_\wedge b$ if $\forall i. a_i \geq b_i$. The other constraints encountered in `add` are resolved similarly.

A more complex example illustrating further type checking challenges is the generalized selection `gsel`. It serves to select not just a particular array element, but an entire subarray. For example, an individual row may be selected from a matrix by accessing the matrix with a selection vector of length one whose value must index within the rows of the matrix. For selection vectors of length zero, `gsel` acts as the identity function, whereas for full-length selection vectors it behaves like the regular selection.

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

Using four index abstractions, the type of `gsel` concisely describes the constraints between the shape of the array and the length and value of the selection vector. The subset notation $\{I \texttt{ in } ..i\}$ allows to specify a strict upper bound on the value of type indices. The implementation exemplifies the three essential structural operations on vectors: the concatenation $a{+\!\!+}b$ appends vector b to vector a. $\texttt{take}(i,v)$ yields a new vector by selecting the first i elements from v, whereas $\texttt{drop}(i,v)$ discards those elements.

Due to their significance for shape-generic array programming, `take`, `drop` and `++` exist as built-in functions and as index terms. This means that the theorem prover must be able to verify constraints containing these three structural operations. As an example, we present the formula that must be verified in order to show that the selection $a\,[v{+\!\!+}x]$ is valid:

$$\begin{array}{l}\forall d.\,\forall l.\, d \geq 0 \wedge l \geq 0 \wedge l < d+1 \Longrightarrow\\ \forall s^d.\,\forall i^l.\,\forall x^{d-l}.\, s \geq_\wedge 0^d \wedge i \geq_\wedge 0^l \wedge i <_\wedge \texttt{take}(l,s) \wedge\\ x \geq_\wedge 0^{d-l} \wedge x <_\wedge \texttt{drop}(l,s) \Rightarrow i{+\!\!+}x \geq_\wedge 0^d \wedge i{+\!\!+}x <_\wedge s.\end{array}$$

3. Conclusion

In this abstract, we have illustrated key ideas of a new system for type-safe array programming. By introducing quarks as a representation of array content, we achieved that every value in the language is an array. To gain type-safety, the system employs a new restricted variant of dependent types to express constraints between array ranks, shapes, and values. Type checking is then carried out by verifying properties in presburger vector arithmetic, an extension of conventional presburger arithmetic to integer vectors of arbitrary length.

Due to the space limitations of this abstract, we omitted some features of the language as well as its operational semantics, the typing rules and a proof of type-safety. In particular, the system presented contains no tuples or dependent pairs. These are essential for representing arrays of arrays and will also be discussed in a full article.

In the future we would like to extend the system towards polymorphic array programming to allow for more programming convenience. Similar to the work done by Xi (Xi and Pfenning 1999), we intend to allow type arguments to be automatically reconstructed if omitted.

References

C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

K.E. Iverson. *A Programming Language*. John Wiley, New York City, New York, USA, 1962.

K.E. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, Canada, 1995.

C. Moler, J. Little, and S. Bangert. *Pro-Matlab User's Guide*. The MathWorks, Cochituate Place, 24 Prime Park Way, Natick, MA, USA, 1987.

B.C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0262162091.

H. Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

H. Xi and F. Pfenning. Dependent Types in Practical Programming. In A. Aiken, editor, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), San Antonio, Texas, USA*, pages 214–227. ACM Press, 1999.