

Assembling Concurrent Programs Correctly from Data-Parallel Program Bricks

Kai Trojahner

Institute of Software Technology and Programming Languages

University of Lübeck, Germany

trojahner@isp.uni-luebeck.de

Abstract

In this position paper, we advocate functional array programming in conjunction with automatic verification to economically harness the power of modern multi-core processors. Array operations often apply scalar operations uniformly to large numbers of array elements. Thus, their execution can be shared between multiple processing units. Using mechanisms such as functional composition, concurrent programs can be conveniently assembled from these basic data-parallel program bricks. To verify array programs, we use a custom type system that takes array shapes into account. The type checker employs a theorem prover to verify constraints on linear expressions. The system rules out large classes of potential run-time errors. Therefore, a well-typed program can dispense with run-time checks. We may even perform extensive code reorganization that aims at eliminating both temporary arrays and synchronization barriers.

1 Introduction

Data-parallel programming is a convenient and simple means to harness the power of modern multi-core processors. Operations that are applied homogeneously to a large number of operands, can be distributed easily over the individual cores. Array programming languages take data-parallelism to its utmost consequence: every value is a multidimensional array, i.e. a vector, a matrix, or, in particular, a scalar. Since array operations affect entire arrays, array programs are highly expressive and contain plenty of data-parallelism. Prominent examples of array languages are APL [6], J [7], MATLAB [8], and SAC [3].

Array operations such as element-wise addition can be applied to any two arrays of numbers as long as both arrays do have exactly the same shape. However, if the arguments don't have the same shape, the semantics of the array addition is undefined. Interpreted array languages like APL, J, and MATLAB feature a large number of built-in operations, each of which implicitly performs the necessary consistency checks on the structural properties of its arguments before execution.

Although deterministic, naïve interpretation has some major drawbacks: Firstly, dynamic checks require time to compute, introducing a fixed overhead. Secondly, the individual execution of basic array operations gives rise to many temporary arrays which increase memory demand of an application and degrade its run-time performance. E.g., an addition of three arrays $A + B + C$ will first store the result of $A + B$ into a temporary array T before computing $T + C$. Although T is not needed in the remainder of the program, it consumes memory and introduces an additional read and an additional write operation per array element. Thirdly, a strictly sequential execution of the individual data-parallel operations introduces synchronization overhead. Finally, beyond performance considerations, dynamic checks have a strong disadvantage: a program, even if it has been running for a long time, may abruptly terminate with an error message.

The pure functional array programming language SAC is a compiled language aimed at high run-time performance and implicit support for concurrent programming [2]. The pure semantics

of SAC allows for both implicitly concurrent execution and extensive code reordering. The SAC compiler applies various optimizations that try to condense simple array operations into more complex ones [9, 4]. This avoids temporary arrays and maximizes the amount of work done in each data-parallel segment. However, to preserve program semantics, the array optimizations can only be applied if the shapes of the participating arrays are known at compile-time. Moreover, SAC still employs dynamic checks to enforce structural constraints [5].

These problems can be overcome by means of computer aided verification. **Qube** [10] is a recent offspring of SAC that aims at entirely static verification of array programs. In **Qube**, all structural constraints are expressed with a type system based on dependent types that takes array shapes into account. Type checking proceeds by verifying constraints on linear integer expressions with the YICES theorem prover [1]. Since all consistency constraints are verified statically, **Qube** programs may entirely dispense with (implicit) dynamic checks. The absence of potential run-time errors may also facilitate the array optimizations employed by the SAC compiler in more general situations. Thus in the context of **Qube**, the correctness of a program can be exploited for both efficiency and concurrency.

2 Assembling Concurrent Array Programs in Qube

In the array programming paradigm, all values are multidimensional arrays. Besides their base types, these are characterized by two properties: rank and shape vector. An array’s *rank* denotes the array’s number of axes. Its *shape vector* contains the extent of each axis. Individual array elements are selected from a multidimensional array with a selection vector whose length equals the array’s rank and whose value must not exceed the array’s shape vector.

The functional array programming language **Qube** allows for the type-safe specification of shape-generic array programs, i.e. programs that operate on arrays with an arbitrary shape vector and even arbitrary rank. Such shape-generic programs may be specified by means of a WITH-loop, a concept originating from SAC. An expression of the form `gen $x < t_1$ with t_2` defines an array of shape t_1 . Each array element is computed by evaluating t_2 in which x has been replaced with the new element’s position vector. The evaluation order of the individual elements is non-deterministic, i.e. the evaluation of the WITH-loop may be shared between multiple processing units. For example, we may specify a shape-generic, concurrent array addition for two arrays A and B of arbitrary rank and shape vector:

$$\text{gen } x < (\text{shape } A) \text{ with } A[x] + B[x].$$

The expression yields an array with the same shape as A ; each of its elements is obtained by adding the corresponding elements from A and B . The expression fails to compute if either A and B don’t have the same number of axes or when some axis of B is shorter the corresponding axis of A .

To rule out such program errors, **Qube** employs types for arrays that describe both the type of the array’s elements as well as its rank and shape vector. In particular, the shape component of a type is itself an expression. This makes array types a variant of dependent types. Unique to our context of programming with multidimensional arrays, we use integer vectors of statically unknown length to index into the family of array types. For example, `[int | [2, 4]]` is the type of integer matrices of shape 2×4 ; the type `[int | vec(n , 2)]` denotes integer arrays of rank n whose axes all have length 2. In order to keep type checking decidable, type-indexes are restricted to a dedicated index language in which only predefined and well-behaved (i.e. linear) operations are permitted. Constraints over these index terms that arise during type checking are resolved with the YICES theorem prover (after eliminating the vector terms in a preprocessing step). A suitable typed variant of the shape-generic array addition captures the constraint that both arguments must have arbitrary but equal ranks and shape vectors:

$$\begin{aligned} \text{add} & : 'r : \text{nat} \rightarrow 's : \text{natvec}(r) \rightarrow [\text{int} | s] \rightarrow [\text{int} | s] \rightarrow [\text{int} | s] \\ \text{add } 'r \ 's \ A \ B & = \text{gen } x < (\text{shape } A) \text{ with } A[x] + B[x] \end{aligned}$$

Here, $'r$ and $'s$ are type-indexes solely needed for type checking; they do not have a run-time representation. Successful type checking proofs that `add` will not abort with a run-time error whenever applied to shape-conforming arguments. Since this property is itself statically verified by the type system, no run-time checks are required to enforce the language semantics.

The function `add` is a correct, data-parallel building block that implements the addition of two integer arrays. We may reuse this functionality in many situations, in the most simple case to define a ternary array addition.

```
add3 : 'r : nat → 's : natvec(r) → [int|s] → [int|s] → [int|s] → [int|s]
add3 'r 's A B C =
  let T = add 'r 's A B in
  add 'r 's T C
```

The ternary array addition `add3` first stores the sum of `A` and `B` into a temporary array `T` before computing the sum of `T` and `C`. The type checker verifies that both applications of `add` are legal and hence no run-time errors will arise during the evaluation of `add3`. However intuitive and correct, the solution is undesirable w.r.t. memory consumption and computing speed. The temporary array `T` consumes memory and introduces an additional read and an additional write operation per array element. Furthermore, the two `WITH`-loops in `add` translate into two consecutive regions of parallel execution, entailing synchronization overhead.

Thus, although composing concurrent programs from small data-parallel building block is desirable from a software-engineering point-of-view, the resulting programs leave a lot to be desired concerning memory and time efficiency. Fortunately, as the program has been proved correct, we may apply rather aggressive optimizations without putting the program semantics at risk. As a first step, we inline the two applications of `add`.

```
add3 : 'r : nat → 's : natvec(r) → [int|s] → [int|s] → [int|s] → [int|s]
add3 'r 's A B C =
  let T = gen x < (shape A) with A[x] + B[x] in
  gen x' < (shape T) with T[x'] + C[x']
```

Inlining the definition of `add` discloses the two `WITH`-loops in the context of `add3`. Since we have proof that every selection into `T` will evaluate successfully, a second transformation replaces the selection `T[x']` with the selected element's definition `A[x'] + B[x']`. Similarly, the transformation may replace the expression computing the result shape `shape T` with `shape A`, eliminating the last reason for computing `T` at all. In the context of SAC, this transformation is known as `WITH`-loop-folding [9]. However, the SAC compiler may only apply `WITH`-loop-folding when all array shapes are known at compile-time.

```
add3 : 'r : nat → 's : natvec(r) → [int|s] → [int|s] → [int|s] → [int|s]
add3 'r 's A B C = gen x' < (shape A) with A[x'] + B[x'] + C[x']
```

The resulting variant of `add` eliminates the shortcomings of the original definition: it computes the result in one conceptual step; no temporary array is needed to store an intermediate result. This decreases memory consumption and improves the program's run-time behaviour. Moreover, the program does not need to perform an intermediate synchronization when evaluated concurrently.

3 Conclusion

In this paper, we have illustrated with a simple example that array programming allows the programmer to conveniently assemble concurrent programs from elementary building blocks. To verify such programs, the functional array programming language `Qube` uses a custom type system based on dependent types that takes array shapes into account. As the type checker employs an SMT solver to verify the necessary constraints, the system feels very much like a type system for a

mainstream programming language that either accepts or rejects a program with an appropriate message.

We have demonstrated how the correctness asserted by the type checker is crucial for the automatic removal of inefficiencies from the assembled array program. Eliminating temporary arrays improves the program's memory and run-time efficiency as well as its communication complexity. However, such transformations are only semantically sound if no run-time errors are eliminated along the way.

We are currently developing a compiler for **Qube** that implements type checking and a basic compilation scheme to OCAML. In the future, the system will exploit the program correctness to generate more efficient programs for both sequential and parallel execution as outlined in this paper.

References

- [1] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.
- [2] C. Grelck. Shared Memory Multiprocessor Support for Functional Array Processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.
- [3] C. Grelck and S.-B. Scholz. SAC: A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.
- [4] C. Grelck, S.-B. Scholz, and K. Trojahner. With-Loop Scalarization: Merging Nested Array Operations. In P. Trinder and G. Michaelson, editors, *Implementation of Functional Languages, 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003, Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 118–134. Springer Verlag, Berlin, Germany, 2004.
- [5] S. Herhut, S.-B. Scholz, R. Bernecky, C. Grelck, and K. Trojahner. From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In O. Chitil, Z. Horváth, and V. Zsóok, editors, *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages (IFL'07), Freiburg, Germany, Revised Selected Papers*, volume (accepted) of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 2008.
- [6] K. Iverson. *A Programming Language*. John Wiley, New York City, New York, USA, 1962.
- [7] K. Iverson. *J Introduction and Dictionary*. Iverson Software Inc., Toronto, Canada, 1995.
- [8] C. Moler, J. Little, and S. Bangert. *Pro-Matlab User's Guide*. The MathWorks, Cochituate Place, 24 Prime Park Way, Natick, MA, USA, 1987.
- [9] S.-B. Scholz. WITH-loop-folding: Condensing Consecutive Array Operations. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK*, pages 225–242. University of St. Andrews, 1997.
- [10] K. Trojahner and C. Grelck. Dependently Typed Array Programs Don't Go Wrong. In E. B. Johnson, O. Owe, and G. Schneider, editors, *Proceedings of the 19th Nordic Workshop on Programming Theory (NWPT'07), Oslo, Norway, October 10-12, 2007*. University of Oslo, Institute of Informatics, Norway, 2007.