Towards Hybrid Array Types in SAC

Grelck, C.; Tang, F.

Publication date
2014
Document Version
Final published version
Published in
SE-WS 2014 : Software Engineering Workshops 2014

Towards Hybrid Array Types in SAC *

Clemens Grelck, Fangyong Tang
Informatics Institute
University of Amsterdam
Science Park 904
1098XH Amsterdam, Netherlands
c.grelck@uva.nl
f.tang@uva.nl

Abstract: Array programming is characterised by a formal calculus of (regular, dense) multidimensional arrays that defines the relationships between structural properties like rank and shape as well as data set sizes. Operations in the array calculus often impose certain constraints on the relationships of values or structural properties of argument arrays and guarantee certain relationships of values or structural properties of argument and result arrays. However, in all existing array programming languages these relationships are rather implicit and are neither used for static correctness guarantees nor for compiler optimisations.

We propose hybrid array types to make implicit relationships between array values, both value-wise and structural, explicit. We exploit the dual nature of such relations, being requirements as well as evidence at the same time, to insert them either way into intermediate code. Aggressive partial evaluation, code optimisation and auxiliary transformations are used to prove as many explicit constraints as possible at compile time. In particular in the presence of separate compilation, however, it is unrealistic to prove all constraints. To avoid the pitfall of dependent types, where it may be hard to have any program accepted by the type system, we use hybrid types and compile unverified constraints to dynamic checks.

1 Introduction

The calculus of multi-dimensional arrays[MJ91] is the common denominator of interpreted array programming languages like APL [Int93], J [Hui92], Nial [Jen89] as well as the compiled functional array language SAC [GS06] (Single Assignment C). The calculus defines the relationships between the *rank* of an array, a scalar natural number that defines the number of axes or dimensions of an array, the *shape* of an array, a vector of natural numbers whose length equals the rank of the array and whose elements define the extent of the array alongside each axis, and last not least the actual data stored in a flat vector, the *ravel* whose length equals the product of the elements of the shape vector.

Many, if not all, operations in the context of this array calculus impose certain constraints

on argument arrays, both structural and value-wise, and guarantee certain relations between arguments and results or in the case of multiple results also between result values, again both structural and value-wise. For example, element-wise extensions of unary scalar operators guarantee that the shape of the result array is the same as the shape of the argument array. Element-wise extensions of binary scalar operators often require the two argument arrays to have equal (or somewhat compatible) shapes and guarantee that the shape of the result array again is the same as that of the argument arrays (or is computed in a certain way from the argument arrays' shapes). Rotation and shifting operations usually preserve the shape of the argument array to be rotated or shifted. Structural operations like take, drop or tile come with rules that determine the shape of the result array based on the shape of one of the arguments (the array) as well as the value of another argument (the take/drop/tile vector), etc, etc.

In interpreted array languages constraints on argument values are checked at runtime prior to each application of one of the built-in array operations. Knowledge about the structural relationships of argument and result values is not used beyond the actual construction of the result array itself. Such relationships are explicit in the documentation of the built-in language primitives and are implicitly derived from these when defining procedures, but there is no opportunity to make such relationships explicit in the code other than as a comment for documentation purposes.

SAC (Single Assignment C)[GS06] is a compiled array programming language that supports shape- and even rank-generic programming. Functions in SAC may accept argument arrays of statically unknown size in a statically unknown number of dimensions. This generic array programming style brings many software engineering benefits, from ease of program development to ample code reuse opportunities.

SAC sets itself apart from interpreted array languages in a variety of aspects. One important aspect is that all basic array operations as sketched out before are not built-in operators with fixed, hard-wired semantics, but rather are defined functions, implemented by means of a powerful and versatile array comprehension construct and provided as part of the SAC standard library. This design has many advantages in terms of maintainability and extendibility, but brings with it that the shapely relationships of argument and result values of these basic operations are just as implicit as they are in the case of any higher level user-defined function.

Our approach consists of four steps:

1. We extend the array type system of SAC by means to express a fairly wide range of relationships between structural properties of argument and result values. These fall into two categories: constraints on the domain of functions and evidence on properties between multiple result values (as supported by SAC) or between result values and argument values.

2. We weave both categories of relationships, constraints and evidence, into the intermediate SAC code such that they are exposed to code optimisation.

3. We apply aggressive partial evaluation, code optimisation and some tailor-made transformations to statically prove as many constraints as possible.

4. At last, we compile all remaining constraints into dynamic checks and remove any evidence from intermediate code without trace.

Whether or not all shape constraints in a program are met is generally undecidable at compile time. Therefore, we name our approach *hybrid array types* following [FFT06] and, like Flanagan, Freund and Tomb, compile unresolved constraints are into runtime checks.

In many examples our approach is surprisingly effective due to the dual nature of our hybrid types. Even if some relationship cannot be proven at compile time, it immediately becomes evidence which often allows us to indeed prove subsequent constraints. Ideally, we end up with some fundamental constraints on the arguments of the functions exposed by some compilation unit and, based on the evidence provided by these constraints, are able to prove all subsequent constraints within the compilation unit.

Our choice to employ compiler code transformations as a technique to resolve constraints has a twofold motivation. Firstly, the SAC compiler is a highly optimising compiler that implements a plethora of different code transformations, which we now reuse for a different purpose. Secondly, we expect a high degree of cross-fertilisation between constraint resolution and code optimisation for the future.

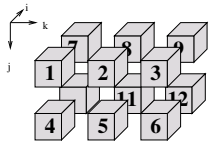
The remainder of the paper is structured as follows. We start with a brief introduction to the array calculus and the type system of SAC in Section 2. In Section 3 we introduce our hybrid types and discuss how they can be inserted into intermediate code in Section 5. Static constraint resolution is demonstrated by means of some examples in Section 6. We discuss some related work in Section 7 and draw conclusions in Section 8.

2 SAC — Single Assignment C

As the name suggests, SAC is a functional language with a C-like syntax. We interpret sequences of assignment statements as cascading let-expressions while branches and loops are nothing but syntactic sugar for conditional expressions and tail-end recursion, respectively. Details can be found in [GS06, Gre12]. The main contribution of SAC, however, is the array support, which we elaborate on in the remainder of this section.

2.1 Array calculus

SAC implements a formal calculus of multidimensional arrays. As illustrated in Fig. 1, an array is represented by a natural number, named the *rank*, a vector of natural numbers, named the *shape vector*, and a vector of whatever data type is stored in the array, named the *data vector*. The rank of an array is another word for the number of dimensions or axes. The elements of the shape vector determine the extent of the array along each of the array's dimensions. Hence, the rank of an array equals the length of that array's shape vector, and the product of the shape vector elements equals the length of the data vector and, thus,



rank: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

rank: 2
shape: [3,3]
data: [1,2,3,4,5,6,7,8,9]

[1, 2, 3, 4, 5, 6]

rank: 1
shape: [6]
data: [1,2,3,4,5,6]

42

rank: 0
shape: [ ]
data: [42]

Figure 1: Calculus of multidimensional arrays

the number of elements of an array. The data vector contains the array's elements in a flat contiguous representation along ascending axes and indices. As shown in Fig. 1, the array calculus nicely extends to "scalars" as rank-zero arrays.

2.2 Array types

The type system of SAC is polymorphic in the structure of arrays, as illustrated in Fig. 2. For each base type (`int` in the example), there is a hierarchy of array types with three levels of varying static information on the shape: on the first level, named AKS, we have complete compile time shape information. On the intermediate AKD level we still know the rank of an array but not its concrete shape. Last not least, the AUD level supports entirely generic arrays for which not even the number of axes is determined at compile time. SAC supports overloading on this subtyping hierarchy, i.e. generic and concrete definitions of the same function may exist side-by-side.

2.3 Array operations

SAC only provides a small set of built-in array operations, essentially to retrieve the rank (`dim(`*array*`)`) or shape (`shape(`*array*`)`) of an array and to select elements or subarrays (*array*`[`*idxvec*`]`). All aggregate array operations are specified using with-loop expressions,

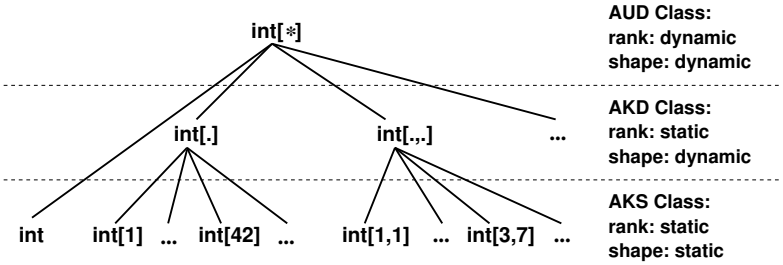


Figure 2: Type hierarchy of SAC

a SAC-specific array comprehension:

```
with {
  ( lower_bound <= idxvec < upper_bound ) : expr;
   ...
  ( lower_bound <= idxvec < upper_bound ) : expr;
}:  genarray( shape, default)
```

This with-loop defines an array of shape `shape` whose elements are by default set to the value of the `default` expression. The body consists of multiple (disjoint) *partitions*. Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to induction variables in for-loops. We call the specification of such an index set a *generator* and associate it with some potentially complex SAC expression that is evaluated for each element of the generator-defined index set.

Based on these with-loops and the support for rank- and shape-invariant programming SAC comes with a comprehensive array library that provides similar functionality as built-in operations in other array languages and beyond. For the SAC programmer these operations are hardly distinguishable from built-ins, but from the language implementor perspective the library approach offers better maintainability and extensibility.

3 Hybrid array types

We introduce a concrete syntax for hybrid array types as shown in Fig. 3. In the long run we envision syntactic sugar to integrate many common hybrid array types directly into the type syntax of SAC as described in the previous section, but as a starting point as well as most general form we introduce hybrid array types in the form of assertions following the C-style function prototype. These assertions make use of a restricted expression syntax as defined in Fig. 3. The existing type inference mechanism is used to ensure that expressions

in assertions are of type `bool`.

Expressions in assertions may contain (integer) constants and variables bound in the (conventional) prototype of the function. Here, we extend the existing SAC (as well as C) syntax and allow names to be given to return values. This is needed to phrase relationships between multiple result values or between arguments and result values.

We deliberately restrict the syntax of constraints as compared to general expressions in SAC. Firstly, we syntactically distinguish between scalar and vector expressions and completely rule out higher dimensional arrays in constraint expression. As usual, one can query for the rank and shape of arrays using the `dim` and `shape` functions, respectively. Furthermore, the usual arithmetic and relational operators are supported both in scalar and vector (1d-array) versions. In addition we make use of a few more functions, well known from the SAC array library (as well as from other array languages):

- `drop( n, vec)` drops the first *n* elements from *vec*;
- `take( n, vec)` takes the first *n* elements from *vec*;
- *vec1* `++` *vec2* concatenates two vectors;
- `tile( n, o, vec)` yields an *n*-element section of *vec* starting at offset *o*.

The restriction of constraint expressions in one way or another is needed to avoid recursive constraint definitions. Consequently, the above functions are technically distinguished from the rank-generic definitions in the SAC standard library that bear the same names.

4 Examples

We illustrate the use of constraints (assertions) to define hybrid array types by means of an example. The original definition of rank-generic element-wise addition (and likewise many other element-wise extensions of scalar operators to arrays) looks as follows:

```
1  float[*] + (float[*] a, float[*] b)
2  {
3    shp = min( shape(a), shape(b));
4    res = with {
5                (0*shp <= iv < shp) : a[iv] + b[iv];
6              }: genarray( shp);
7    return res;
8  }
```

In the presence of a rank- and shape-generic specification, the existing SAC type system has no means to guarantee shape equality or compatibility. To avoid out-of-bound array indexing into either argument array we define the result array to have a shape that is the minimum of the shapes of the two argument arrays, more precisely to have a rank equal

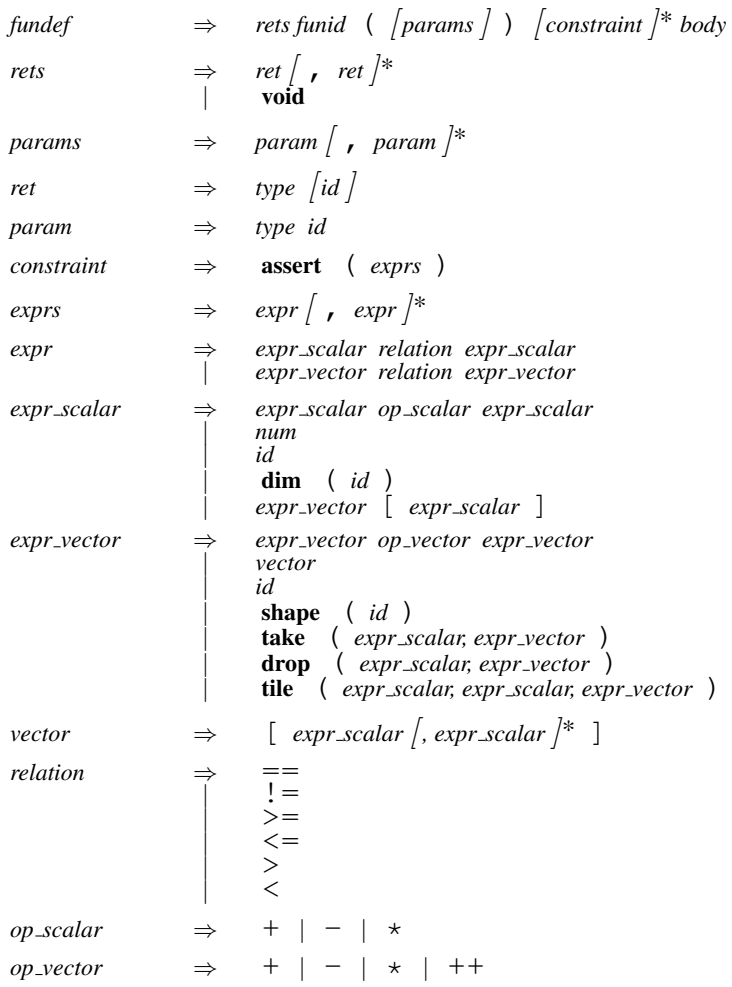


Figure 3: Syntax of user-defined constraints of SAC

to the lesser rank of the argument arrays and for each axis within this rank the minimum number of elements of either argument array. This is save, but has a few drawbacks.

In many cases adding two arrays of different shape must rather be considered a programming error. The above definition disguises this programming error until it hits back at a different location. Also from a compilation point of view the above solution is suboptimal. If we knew at compile time that both argument arrays were of the same shape and thus likewise the produced result, we could immediately reuse either argument array's storage space (if no longer needed outside the current context) to store the result array and thus significantly reduce the memory footprint of the operation [GT04]. With the new hybrid array types we have the opportunity to exactly express this:

```
1  float[*] c + (float[*] a, float[*] b)
2  assert( shape(a) == shape(b))
3  assert( shape(c) == shape(a))
4  {
5    shp = shape(a);
6    res = with {
7                (0*shp <= iv < shp) : a[iv] + b[iv];
8              }: genarray( shp);
9    return res;
10 }
```

Note that the name of the return value in the function prototype can well be different from the name of the variable in the return-statement in the function body.

Note further that we use the same key word `assert` to define different kinds of type restrictions. Assertions that exclusively refer to argument identifiers define constraints on the function's domain. They are similar to preconditions. In contrast, any assertion that includes to identifiers denoting result values defines additional static knowledge on result values. They resemble postconditions.

5 Turning type assertions into guards and evidence

In order to use existing optimisation capabilities for the static resolution of hybrid array types we must represent the assertions within the intermediate SAC code in the right way. We illustrate this by means of an example:

```
1  float[*] c, float[*] d plusminus (float[*] a, float[*] b)
2  assert( shape(a) == shape(b))
3  assert( shape(c) == shape(d))
4  assert( shape(c) == shape(a))
5  {
6    return (a+b, a-b);
7  }
```

The function `plusminus` yields the element-wise sum and the element-wise difference of two arbitrarily shaped arguments; its definition makes use of the element-wise sum function introduced in the previous section and a similarly defined difference function. The shapes of all four arrays involved must be equal as is expressed by three assertions. The first assertion exclusively refers to arguments. Thus, it defines a precondition or domain restriction. The other two assertions involve either exclusively or partially result values and hence define postconditions.

Our goal is to represent the various constraints in such a way that their dual nature as guards as well as knowledge is properly exposed. If successful, we expect to statically prove the preconditions of the sum and difference functions based on the precondition

of the plusminus function. Moreover, we expect to statically prove the postconditions of the plusminus function based on the postconditions of the sum and difference functions and last not least to prove their postconditions based on their definitions and the shapely properties of the with-loops involved.

We use 3 internal primitives to represent constraints in the intermediate SSA-based representation of SAC code: `guard`, `evidence` and `collect` as illustrated by means of the `plusminus` function:

```
1  float[*] c, float[*] d plusminus (float[*] a, float[*] b)
2  {
3    g1 = evidence( shape(a) == shape(b));
4    t1 = a+b;
5    t2 = a-b;
6    g2 = guard( shape(t1) == shape(t2);
7    g3 = guard( shape(t1) == shape(a);
8    t3, t4 = collect( t1, t2, g1, g2, g3);
9    return (t3, t4);
10 }
```

The `guard` primitive, if not resolved statically, represents a classical assertion: it checks the condition and terminates program execution if the predicate is not met. As the flip side of the coin, a `guard` also ensures that condition holds; this additional knowledge can be exploited by the compiler. The `evidence` primitive merely represents the second aspect: representation of knowledge. Last not least, in a functional, purely dataflow-oriented context (despite the C-like syntax used), we must weave all occurrences of either primitive into the dataflow. In order to do so without imposing barriers for conventional optimisation we add the `collect` primitive that always presides the return-statement and *collects* the various results of constraints and evidence primitives in order to make the *needed* in the function's data flow.

For the sum function the corresponding transformation yields the following intermediate representation:

```
1  float[*] c + (float[*] a, float[*] b)
2  {
3    g1 = evidence( shape(a) == shape(b));
4    shp = shape(a);
5    res = with {
6               (0*shp <= iv < shp) : a[iv] + b[iv];
7            }: genarray( shp);
8    g2 = guard( shape(res) == shape(a));
9    res2 = collect( res, g1, g2);
10   return res2;
11 }
```

Why do we represent the precondition of the `plusminus` function with `evidence` and not with `guard`? Assuming `plusminus` is an exported symbol of some compilation

unit, where would the necessary dynamic check come from?

This is exactly why we lift the whole issue on the type level rather than adding explicit assertions to the language. There is no way to prove a precondition in the function body. As a consequence any precondition inflicts a `guard` in the calling context. If we still evaluate the function definition we know for sure that the predicate holds, hence the `evidence` in the function body.

For postconditions it works exactly the other way around. We add `guard` into the function definition because only in the definition do we stand a chance to prove the predicate. If we return to the calling context, we thus know that the predicate holds: it becomes `evidence` in the calling context. We demonstrate this pair-wise occurrence of `guard` and `evidence` by completing our example:

```
1  float[*] c + (float[*] a, float[*] b)
2  {
3    g1 = evidence( shape(a) == shape(b));
4    shp = shape(a);
5    res = with {
6               (0*shp <= iv < shp) : a[iv] + b[iv];
7            }: genarray( shp, 0);
8    g2 = evidence( shape(res) == shp ++ shape(0));
9    g3 = guard( shape(res) == shape(a));
10   res2 = collect( res, g1, g2,g3);
11   return res2;
12 }
13
14 float[*] c, float[*] d plusminus (float[*] a, float[*] b)
15 {
16   g1 = evidence( shape(a) == shape(b));
17   g2 = guard( shape(a) == shape(b));
18   t1 = a+b;
19   g3 = evidence( shape(t1) == shape(a));
20   g4 = guard( shape(a) == shape(b));
21   t2 = a-b;
22   g5 = evidence( shape(t2) == shape(a));
23   g6 = guard( shape(t1) == shape(t2);
24   g7 = guard( shape(t1) == shape(a);
25   t3, t4 = collect( t1, t2, g1, g2, g3, g4, g5, g6,g7);
26   return (t3, t4);
27 }
```

In the definition of the element-wise sum function we also add the evidence that arises from the semantics of the with-loop (line 8): The result shape is the concatenation of the with-loop's shape vector (first expression position after the `genarray` key word) and the so-called *element shape* as determined by the with-loop's default element (second expression position after the `genarray` key word).

We omit the definition of the element-wise difference function as it is completely analogous to the element-wise sum function.

6 Resolving hybrid types

Our goal is to prove as many shape constraints at compile time as possible. For this we reuse the aggressive optimisation capabilities of the SAC compiler now that we have properly represented type constraints within the intermediate SAC code. In a first step we apply common subexpression elimination to a number of identical predicates. Furthermore, we simplify evidence g2 in the definition of the sum function: the shape of a scalar constant is the empty vector, which is the neutral element of vector concatenation. This first round of optimisation yields the following code representation:

```
1  float[*] c + (float[*] a, float[*] b)
2  {
3    p1 = shape(a) == shape(b);
4    g1 = evidence( p1);
5    shp = shape(a);
6    res = with { (0*shp <= iv < shp) : a[iv] + b[iv];
7          }: genarray( shp, 0);
8    p2 = shape(res) == shp;
9    g2 = evidence( p2);
10   p3 = shape(res) == shape(a);
11   g3 = guard( p3);
12   res2 = collect( res, g1, g2,g3);
13   return res2;
14 }
15
16 float[*] c, float[*] d plusminus (float[*] a, float[*] b)
17 {
18   p1 = shape(a) == shape(b);
19   g1 = evidence( p1);
20   g2 = guard( p1);
21   t1 = a+b;
22   p2 = shape(t1) == shape(a);
23   g3 = evidence( p2);
24   g4 = guard( p1);
25   t2 = a-b;
26   p3 = shape(t2) == shape(a);
27   g5 = evidence( p3);
28   p4 = shape(t1) == shape(t2);
29   g6 = guard( p4);
30   g7 = guard( p2);
31   t3, t4 = collect( t1, t2, g1, g2, g3, g4, g5, g6,g7);
32   return (t3, t4);
33 }
```

A number of predicates in plusminus appear both in guard and evidence positions. The evidence ensures us that the predicate holds, hence the corresponding guards can be removed. The same holds for guard g3 in the sum function after one more round of variable propagation and common subexpression elimination. The result looks as follows:

```
1  float[*] c + (float[*] a, float[*] b)
2  {
3    p1 = shape(a) == shape(b);
4    g1 = evidence( p1);
5    res = with { (0*shape(a) <= iv < shape(a)) : a[iv] + b[iv];
6          }: genarray( shape(a), 0);
7    p2 = shape(res) == shape(a);
8    g2 = evidence( p2);
9    res2 = collect( res, g1, g2);
10   return res2;
11 }
12
13 float[*] c, float[*] d plusminus (float[*] a, float[*] b)
14 {
15   p1 = shape(a) == shape(b);
16   g1 = evidence( p1);
17   t1 = a+b;
18   p2 = shape(t1) == shape(a);
19   g3 = evidence( p2);
20   t2 = a-b;
21   p3 = shape(t2) == shape(a);
22   g5 = evidence( p3);
23   p4 = shape(t1) == shape(t2);
24   g6 = guard( p4);
25   t3, t4 = collect( t1, t2, g1, g3, g5, g6);
26   return (t3, t4);
27 }
```

With all guards resolved in the definition of sum (and analogously difference) we can now focus on plusminus. One more round of common subexpression elimination yields:

```
1  float[*] c, float[*] d plusminus (float[*] a, float[*] b)
2  {
3    as = shape(a);
4    bs = shape(b);
5    p1 = as == bs;
6    g1 = evidence( p1);
7    t1 = a+b;
8    t1s = shape(t1);
9    p2 = t1s == as;
10   g3 = evidence( p2);
11   t2 = a-b;
12   t2s = shape(t2);
13   p3 = t2s == as;
```

```
14    g5 = evidence( p3);
15    p4 = t1s == t2s;
16    g6 = guard( p4);
17    t3, t4 = collect( t1, t2, g1, g3, g5, g6);
18    return (t3, t4);
19  }
```

Next, we exploit equality evidence (g1, g3, g5). If two values are equal, we can replace all occurrences of one by the other, making the left operand the representative of the corresponding equivalence class. This yields:

```
1   float[*] c, float[*] d plusminus (float[*] a, float[*] b)
2   {
3     as = shape(a);
4     bs = shape(b);
5     p1 = as == bs;
6     g1 = evidence( p1);
7     t1 = a+b;
8     t1s = shape(t1);
9     p2 = t1s == as;
10    g3 = evidence( p2);
11    t2 = a-b;
12    t2s = shape(t2);
13    p3 = t2s == t1s;
14    g5 = evidence( p3);
15    p4 = t2s == t2s;
16    g6 = guard( p4);
17    t3, t4 = collect( t1, t2, g1, g3, g5, g6);
18    return (t3, t4);
19  }
```

The last remaining guard (g6) is resolved through reflexivity of the equivalence relation equality, a standard compiler optimisation. As a final step we remove all occurrences of evidence and run a final round of dead code removal and variable propagation, which results in

```
1   float[*] c, float[*] d plusminus (float[*] a, float[*] b)
2   {
3     t1 = a+b;
4     t2 = a-b;
5     return (t1, t2);
6   }
```

In essence, we managed to resolve all type constraints within the definition of plusminus including all constraints in the locally defined functions sum and difference. Assuming plusminus is exported from the compilation unit under consideration, its hybrid type ensures that applications of plusminus are protected by a guard ensuring that both argument

arrays are of the same shape. Within the calling context this guard is subject to similar compiler transformations as just shown and potentially this guard may also be removed.

7 Related work

The term *hybrid type* was coined in the area of object-oriented programming [FFT06]. Similar to that approach we aim at statically resolving as many type constraints as possible, but we accept that it is generally undecidable to resolve all such constraints at compile time and thus we leave the remaining ones to dynamic checks rather than rejecting the program as ill-typed. Our work differs form the above referenced by the deliberate restriction to shapely relationships within the calculus of multidimensional arrays, which we expect to give us much better chances for static resolution than would be found in the general case.

In practice, our approach is not entirely dissimilar to preconditions and postconditions as (arguably) first proposed in the *The Vienna Development Method (VDM)* [BJ78]. VDM is/was first and foremost a specification method and not meant to statically resolve constraints as we expect nor is/was it in any way related to arrays and their specific relationships.

Another related approach is *design by contract* proposed by Bertrand Meyer [Mey91, Mey92], which is widely used both in object-oriented programming languages [BFMW01, Plo97] and functional programming languages [Xu06, FF02, BM06]. But none of them has concrete shape restrictions on arrays as described in this paper.

Interpreted array programming languages like APL [Ive62] and J [Ive95] are dynamically typed. When the interpreter encounters an array operation, it first checks whether the arguments properly match in rank and shape regarding the specific semantic requirements of the operation. If so, the interpreter performs the corresponding computation; if not, it aborts program execution with an error message. The drawback of any interpretive approach is twofold: any programming error can only materialise at runtime, and repeatedly checking argument properties can have a considerable runtime impact [Ber98]. Our proposed compiler technology counters both aspects by giving programmers compile time information, e.g. including hints where and why dynamic checks do remain in the code, and by only leaving those dynamic checks in the code that cannot be resolved at compile time.

Our work indeed is very related to Qube [Tro11, TG09], a programming language that combines the expressiveness of array programming similar to SAC with the power of fully-fledged dependent array types. The difference to our current work is that the Qube compiler rejects any program for which it cannot prove type-correctness. In practice, this means that, just as with other dependently typed programming languages, writing a type-correct program can be challenging. In contrast, with our proposed hybrid type approach we will in practice accept a much wider variety of programs.

Further related work in the context of SAC focusses on checking domain constraints of SAC's built-in functions [HSB+08]. In contrast, in our current work we target general user-defined functions and, thus, more general constraints. To resolve these constraints,

among others, we adopt *symbiotic expressions* [BHS11], which is a method for algebraic simplification within a compiler.

8 Conclusion

We propose hybrid array types as a refinement of standard SAC array types by user-defined constraints that allow us to more accurately specify the shapely relationships between parameters and return values of functions. Shapely constraints involving multiple parameters can be used to restrict the domain of functions while constraints between multiple return values or constraints between parameters and return values provide the compiler with additional information on the code, which can be exploited for general code optimization just as for the static resolution of constraints.

We aim at resolving as many constraints as possible at compile time and only to leave the remaining ones optionally as runtime checks. Here, we exploit the dual nature of such runtime checks: runtime checks in the first place, but likewise static evidence of the checked property in the remaining code. As we demonstrated by means of a small case study, it is typical that a few constraints remain as dynamic assertions in the code due to lack of information on argument values and separate compilation, but with their evidence many (often all) subsequent constraints within a compilation unit can statically be resolved.

This resolution is mainly based on aggressive partial evaluation and code optimization, characteristic for the SAC compiler. Nonetheless, we identified a number of additional code transformations that one by one may be considered trivial, but which in conjunction make the more sophisticated optimizations, namely common subexpression elimination, considerably more effective, both in general optimization as well as in constraint resolution. Following the credo *as much static resolution as possible, as many dynamic checks as necessary* our approach may, however, prove simpler for programmers as the inability to statically solve all constraints does not let the compiler reject a program as illegal.

We are currently busy implementing hybrid types with shape constraints in the SAC compiler and look forward to gather more experience with the approach across a wide range of applications. Issues of interest are among others questions about the practicability of the approach, both with respect to user acceptance as well as compilation times and size of intermediate code due to the addition of constraint representations.

References

[Ber98] R. Bernecky. Reducing computational complexity with array predicates. In *ACM SIGAPL APL Quote Quad*, pages 39–43. ACM, 1998.

[BFMW01] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass–Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[BHS11] R. Bernecky, S. Herhut, and S.B. Scholz. Symbiotic expressions. *Implementation and Application of Functional Languages*, pages 107–124, 2011.

[BJ78] Dines Bjørner and Cliff B. Jones. The Vienna Development Method: The Meta-Language. In *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[BM06] M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4-5):375–414, 2006.

[FF02] R.B. Findler and M. Felleisen. Contracts for higher-order functions. *ACM SIGPLAN Notices*, 37(9):48–59, 2002.

[FFT06] Cormac Flanagan, Stephen N Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. *FOOL/WOOD*, 6, 2006.

[Gre12] C. Grelck. Single Assignment C (SAC): High Productivity meets High Performance. In V. Zsok Z. Horvath, editor, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*. Springer, 2012. to appear.

[GS06] 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.

[GT04] Clemens Grelck and Kai Trojahner. Implicit Memory Management for SaC. In Clemens Grelck and Frank Huch, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, pages 335–348. University of Kiel, Institute of Computer Science and Applied Mathematics, 2004. Technical Report 0408.

[HSB+08] Stephan Herhut, Sven-Bodo Scholz, Robert Bernecky, Clemens Grelck, and Kai Trojahner. From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In Olaf Chitil, Zoltan Horváth, and Viktória Zsók, editors, *19th International Symposium on Implementation and Application of Functional Languages (IFL'07), Freiburg, Germany, Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2008.

[Hui92] R.K.W Hui. *An Implementation of J*. Iverson Software Inc., Toronto, Canada, 1992.

[Int93] International Standards Organization. Programming Language APL, Extended. ISO N93.03, ISO, 1993.

[Ive62] K.E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 345–351. ACM, 1962.

[Ive95] K.E. Iverson. *J Introduction and Dictionary*. New York, NY, USA, 1995.

[Jen89] M.A. Jenkins. Q'Nial: A Portable Interpreter for the Nested Interactive Array Language Nial. *Software Practice and Experience*, 19(2):111–126, 1989.

[Mey91] B. Meyer. EIFFEL: The language and environment. *Prentice Hall Press, 300*, 1991.

[Mey92] B. Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.

[MJ91] L.M. Restifo Mullin and M. Jenkins. A Comparison of Array Theory and a Mathematics of Arrays. In *Arrays, Functional Languages and Parallel Systems*, pages 237–269. Kluwer Academic Publishers, 1991.

[Plo97] R. Plosch. Design by contract for Python. In *Software Engineering Conference, 1997. Asia Pacific... and International Computer Science Conference 1997. APSEC'97 and ICSC'97. Proceedings*, pages 213–219. IEEE, 1997.

[TG09] K. Trojahner and C. Grelck. Dependently typed array programs don't go wrong. *Journal of Logic and Algebraic Programming*, 78(7):643–664, 2009.

[Tro11] K. Trojahner. *QUBE–Array Programming with Dependent Types*. PhD thesis, University of Lübeck, Lübeck, Germany, 2011.

[Xu06] Dana N. Xu. Extended static checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 48–59, New York, NY, USA, 2006. ACM.