User-Defined Shape Constraints in SAC

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

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

Abstract

We propose a method called user-defined constraints specifically for shape-generic multi-dimensional array programming. Our proposed technique allows programmers to make implicit constraints in the domain and codomain of functions explicit. This method can help compilers to generate more reliable code, improve performance through better optimization and improve software documentation.

We propose and motivate a syntax extension for the functional array language SAC and describe steps to systematically transform source-level constraints into existing intermediate code representations. We discuss ways of statically resolving constraints through aggressive partial evaluation and propose some form of syntactic sugar that blurs the line between user-defined constraints and fully-fledged dependent types.

1. Introduction

SAC (Single Assignment C) is an array programming language that supports shape-generic programming[1], i.e., functions may accept argument arrays with 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.

However, generic array programming also introduces some subtle pitfalls. Many array operations are characterized by implicit shape constraints on parameters or return values. For example, matrix multiplication requires the second axis of the first parameter to be as long as the first axis of the second parameter; in element-wise arithmetic (e.g. sums and products of two arrays) it is often desirable to ask for shape equality of arguments. If programmers do not express implicit constraints in the code, merely depending on inferring shape information of parameters by compiler (which is really hard) is not enough to generate reliable executable code. Insufficient information about relationship of function parameters and result values is a problem. Program execution may encounter runtime errors, e.g. out-of-bound array indexing, if implicit constraints are violated, or program execution may simply yield erroneous results.

How to express constraints of functions is a question. For instance, programmers may add conditional code around function applications or in the definition of the applied function. However, both approaches have their disadvantages. In the first case, programmers write redundant code for each function application, which violates software engineering principles. In the second case, compilers have little opportunity to prove constraints, and most constraints have to be left for dynamic checks. The lack of distinction between functionally relevant and constraint checking code in both scenarios further complicates a compiler's job.

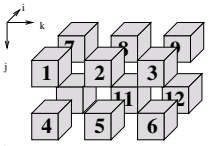
Therefore, it is useful if programmers can explicitly supply constraints in code. We propose user-defined constraints that explicitly annotate relations between parameters and/or return values of functions. This approach facilitates program error detection and helps to infer more precise type information for smooth optimization. Using this approach, we obtain the following benefits:

1. User-defined constraints can help a compiler to generate more reliable code. For instance, a generic function *vector add* takes two vectors of the same size as parameters and yields a new vector whose values are the sums of the corresponding elements of the two argument vectors. In the absence of shape constraints the only way to avoid potential out-of-bound indexing into one of the argument vectors is to generate a vector whose length equals that of the shorter of the two argument vectors. With the proposed user-defined shape constraints, we instead could explicitly restrict the domain of the *vector add* function.
2. Using annotated code can improve performance through better optimization. If the constraints can be statically resolved, corresponding code will be removed, and resolved constraints can be propagated to facilitate further optimization.
3. Showing constraints of function, to some extent, can represent software documentation. Since relationships of return values or parameters are given, it helps programmers to better understand code, e.g. generic function definition of matrix multiplication with user-defined constraints.

The main contributions of this paper are:

1. a new method called user-defined constraints that explicitly express constraints of functions;
2. an outline of syntax of the innovative method;
3. a discussion about where and how to assert the constraints;
4. a practical case is given to show how this approach works.

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. Section 3 presents the motivation of the proposed method. A detailed description and syntax of user-defined constraints is given



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

in Section 4. Section 5 describes code transformation of constraints, and Section 6 discusses where and how to insert constraints into intermediate code. We sketch out some syntactic sugar for shape constraints in Section 7. Related work is discussed in Section 8. Finally, we draw conclusions in Section 9.

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 [1, 2]. 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, 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.

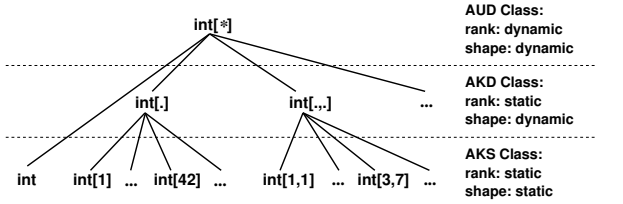


Figure 2. Type hierarchy of SAC

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, 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. Motivation

In this section, we use matrix multiplication ("matmul" for short) as example to show how to define a function on different levels of abstraction according to the hierarchy of array types of SAC. We demonstrate the existence of implicit constraints and motivate the desire for explicit constraints.

```
1  float[3,6] matmul(float[3,5] A, float[5,6] B)
2  {
3    BT = transpose(B);
4    C = with{
5          ([0,0]<=[i,j]<[3,6]):sum(A[i]*BT[j]);
6        }:genarray([3,6],0f);
7    return(C);
8  }
```

Figure 3. Function definition of matrix multiplication with AKS type

3.1 AKS function definition

Fig. 3 shows how to define matrix multiplication for arrays of fixed shapes. We first transpose array B (line 3) before we use a single with-loop for the standard textbook definition of matrix multiplication (line 4-7).

Even though it seems that this function definition reveals the well-known shape constraints on function parameters, it merely does so for the concrete argument shapes, but it does not reveal the underlying general relationship. What's more, this code only works for one pair of array shapes, which very much limits code reuse. Therefore, a more generic solution is needed.

3.2 AKD function definition

As introduced in Section 2, an AKD type, e.g. `float[.,.]`, defines the number of axes, but leaves the extent along each axis open. Matrix multiplication can be written in a shape-generic way, as shown in Fig. 4.

```
1  float[.,.] matmul(float[.,.] A,float[.,.] B)
2  {
3    BT = transpose(B);
4    a0 = shape(A)[0];
5    b1 = shape(B)[1];
6    C = with{
7        ([0,0]<=[i,j]<[a0,b1]):sum(A[i]*BT[j]);
8        }: genarray([a0,b1],0f);
9    return(C);
10 }
```

Figure 4. Function definition of matrix multiplication with AKD type

In this generic code, we did not supply constraints for function. What happens if we apply the function to wrongly shaped arguments? We give mismatched matrices for this function, for instance, two arrays with shape of [3,5] and [6,3], respectively. Element-wise vector multiplication (e.g. `A[i]*BT[j]`) is defined in the SAC standard library. In the absence of shape constraints it is defined to yield a vector whose length equals the minimum of the lengths of the argument vectors. This way, out-of-bound array indexing can be avoided at the expense of an additional minimum computation and obfuscation of the shape relationships, which is detrimental for further optimization. For our definition of matrix multiplication this means that we avoid a runtime error and make function `matmul` total, but we do this outside the mathematical definition of matrix product.

3.3 AUD function definition

Array programs are composed from general-purpose array operations, and many operations are rank-generic, i.e. they are applicable to arrays with arbitrary number of axes. AUD types like `float[*]` encompass all arrays of a given base type regardless of their structure. Here we generalise *matmul* to *innerproduct*. As shown in Fig. 5, function `innerproduct` takes two arrays `A` and `B`, where the last axis of A has to be as long as the first axis of B.

```
1  float[*] innerproduct(float[*] A,float[*] B)
2  {
3    al = dim(A)-1;
4    bl = dim(B)-1;
5    a = drop([-1],shape(A));
6    b = drop([1],shape(B));
7    v = a++b;
8    C = with{
9          (0*v<=iv<v) {
10           m = take([al],iv);
11           n = take([-bl],iv);
12           var = sum(take(m,A)*take(n,B));
13         }:var;
14       }:genarray(v,0f);
15   return(C);
16 }
```

Figure 5. Function definition of inner product with AUD type

Values `al` and `bl` are computed and used in line 10-11 for taking specific axis. Line 5-6 get first `al` and last `bl` axis from two shape of parameters respectively by using `drop` function for simple, which comprise shape of final result in line 7. Line 8-13 do inner product computation with with-loop.

Functions `take(sv,a)` and `drop(sv,a)` take and drop as many elements from array `a` as indicated by shape vector `sv`, respectively. Each element of `sv` corresponds to one axis of `a` starting from the leftmost one. For positive components of `sv`, the elements are taken or dropped from the "beginning", i.e., starting with index 0. Otherwise, they are taken or dropped from the "end", i.e., starting from the maximum legal index of the corresponding axis.

3.4 Implicit constraints

Program errors are common in software systems and often hard to detect, especially the implicit constraints discussed above, which may cause runtime errors, e.g. out-of-bound array access. What's more, such constraints are enforced by means of dynamic checks that carry a performance penalty. To avoid this problem, programmers could add additional conditional statements either in the callee or in the caller function, but none of them can express constraints in function level. In the next section, we introduce a user-defined constraints technology, which can express constraints explicitly. Through existing aggressive partial evaluation in SAC, these user-defined constraints may be resolved and removed by compiler to avoid runtime checks.

4. Proposed user-defined constraints

The constraints in program vary, which not only focus on parameter, but also return value in generic functions. In this section, a compiler technology called user-defined constraints is introduced. This technique allows programmers to express constraints of functions explicitly. While still writing code in a generic way, programmers can add their desired annotations between function declaration and function body, which indicates compiler constraints on parameters and/or return value without losing generalization.

According to different usage of constraints, the technology can be divided into two parts:

1. Precondition (user-defined constraints on parameters): programmers can specify the limitations of parameters using dedicated expression;
2. Postcondition (user-defined constraints on return values): programmers can also add constraints on shape or rank information of return value.

We introduce two key words for SAC: `where` is used to express the constraints between function parameters while `assert` is used to indicate shape or rank information of return values.

4.1 User-defined constraints on parameters

In this part, we use `where` to express constraints between parameters. For function in Fig. 4, because the ranks of parameters are known and equal, programmers just need to restrict specific axes of parameters, as demonstrated in Fig. 6. Using this method, the function `matmul` documents the constraint clearly: the second axis of the first parameter has to be as long as the first axis of the second parameter.

```
1  float[.,.] matmul(float[.,.] A,float[.,.] B)
2  where(shape(A)[1]==shape(B)[0])
3  {
4    /*Computation*/
5  }
```

Figure 6. Constraint on dedicated axis of parameters in AKD type

Fig. 6 shows basic usage of user-defined constraints on specific axes of parameters. There are some other abstract restrictions of function parameters. For example, the generic array addition function in Fig. 7 requires the shape of array A to be equal to the shape of array B. Just using the above method — express axis-wise equality — would be tedious and even impossible for AUD-style functions. Therefore, we directly use the equality of shape vectors of parameters to express this constraint, as shown in Fig. 7.

```
1  int[*] addition(int[*] A, int[*] B)
2  where(shape(A)==shape(B))
3  {
4    /*Computation*/
5  }
```

Figure 7. Constraint on shape of parameters

In Fig. 6, == is used to represent equality of two scalar values, however, here, vectors appear as operands. In SAC, == is more general operation which can indicate the equality not only between scalars but also vectors and arrays. How == will be transformed will be illustrated in next section. `shape(A)==shape(B)` means shape of parameters A and B are the same. At the same time, it indicates that the ranks of the two array have to be equal as well, which is an implicit constraint inflicted by the user-defined constraint.

Compared with matrix multiplication, our function `innerproduct` in Fig. 5 has a similar restriction: the last axis of the first parameter and the first axis of the second parameter must coincide in length. The built-in function `take` is used to retrieve elements from the shape vector. We can extend builtin functions in `where` domain as well, as illustrated in Fig. 8.

```
1  float[*] innerproduct(float[*] A,float[*] B)
2  where(take([-1],shape(A))
3             ==take([1],shape(B)))
4  {
5    /*Computation*/
6  }
```

Figure 8. Constraints on dedicated axis of parameter in AUD type

Some applications may have other kinds of restrictions among parameters. For instance, function `specialfun` in Fig. 9 has a constraint that from the second element of the shapes of arrays A and B to the sixth, each element must be equal. These constraints can be expressed as in Fig. 9. The function `tile(sv,ov,a)` takes a *tile* of shape `sv` from `a` starting at offset `ov` from array `a`.

```
1  int[*] specialfun(int[*] A, int[*] B)
2  where(tile([2],[4],shape(A))
3           ==tile([2],[4],shape(B)))
4  {
5    /*Computation*/
6  }
```

Figure 9. Constraints on arbitrary shape domain

4.2 User-defined constraints on return values

In some cases, programmers may also want to restrict shapes or ranks of return values or express shapely relationships between multiple return values. Such information can prove indispensable to resolve further constraints later in the code. To express constraints involving return values of functions we need a way to refer to return values in constraint expressions. As shown in Fig. 10, function `addsub` has a comma-separated list of return types in front of the function name and return-statement in the function definition likewise contains a comma-separated list of expressions. At first glance, we could be tempted to use variables occurring in the return-statement. However, in practice we must be able to express constraints without having access to the complete function definition. To solve both issues, we extend the syntax of function definitions to explicitly name return values, just as function parameters.

Constraints among return values For generic functions, like `addsub` in Fig. 10, shape and rank of parameters and return values are all unknown. `assert` expression is used to indicate that the shapes of the two return values should be equal.

```
1  int[*] x,int[*] y addsub(int[*] A,int[*] B)
2  where(shape(A)==shape(B))
3  assert(shape(x)==shape(y))
4  {
5    x = A+B;
6    y = A-B;
7    return(x,y);
8  }
```

Figure 10. Constraints among return values

Constraints between parameters and return values

In practice, most return values have some shape relationship with parameters. Take matrix multiplication as an example. The shape of return value contains two elements. The first element is equal to the first axis of the first parameter, and the second element is equal to the second axis of the second parameters. This relationship can be represented as demonstrated in Fig. 11.

```
1  float[.,.] x matmul(float[.,.] A,float[.,.] B)
2  where(shape(A)[1]==shape(B)[0])
3  assert(shape(x)[0]==shape(A)[0],
4         shape(x)[1]==shape(B)[1])
5  {
6    /*Computation*/
7  }
```

Figure 11. Constraints between parameters and return values on dedicated axis

4.3 Syntax of user-defined constraints

We introduce a concrete syntax for user-defined constraints as shown in Fig. 12, +, -, * is extended in SAC to be used in scalar and array operations. Primitive functions `shape`, `dim`, `tile`, `drop`, `take` and `concatenate` are used in this syntax. We simplify the syntax and leave out irrelevant SAC features.

$$
\begin{array}{lcl}
fundef & \Rightarrow & rets\ funid\ (\ args\ )\ \left[\ constraint\ \right] body \\
rets & \Rightarrow & \left(\ ret\left[\ ,ret\ \right]^{*} \mid \left[\ \textbf{void}\ \right]\right) \\
args & \Rightarrow & \left(\ param\left[\ ,param\ \right]^{*} \mid \left[\ \textbf{void}\ \right]\right) \\
ret & \Rightarrow & param \\
 & \mid & type \\
param & \Rightarrow & type\ id \\
constraint & \Rightarrow & \left[\ \textbf{where}\ (\ exprs\ )\ \right]\left[\ \textbf{assert}\ (\ exprs\ )\ \right] \\
exprs & \Rightarrow & expr\left[\ ,expr\ \right]^{*} \\
expr & \Rightarrow & info\_scalar\ equality\ expr\_scalar \\
 & \mid & info\_vector\ equality\ expr\_vector \\
info\_scalar & \Rightarrow & \textbf{dim}\ (\ id\ ) \\
 & \mid & \textbf{shape}\ (\ id\ )\ [const] \\
info\_vector & \Rightarrow & \textbf{shape}\ (\ id\ ) \\
 & \mid & \textbf{take}\ (\ expr\_vector, shape\ (\ id\ )\ ) \\
 & \mid & \textbf{drop}\ (\ expr\_vector, shape\ (\ id\ )\ ) \\
 & \mid & \textbf{tile}\ (\ expr\_vector, expr\_vector, \textbf{shape}\ (\ id\ )\ ) \\
expr\_scalar & \Rightarrow & num\ op\ info\_scalar \\
 & \mid & info\_scalar \\
 & \mid & num \\
 & \mid & id \\
expr\_vector & \Rightarrow & num\ op\ info\_vector \\
 & \mid & info\_vector \\
 & \mid & concatenate\ (\ expr\_vector, expr\_vector\ ) \\
 & \mid & array \\
array & \Rightarrow & [expr\_scalar\left[\ ,expr\_scalar\ \right]^{*}] \\
equality & \Rightarrow & == \\
op & \Rightarrow & + \\
 & \mid & - \\
 & \mid & *
\end{array}
$$

Figure 12. Syntax of user-defined constraints of SAC

5. Transformation of user-defined constraints

Implicit constraints can be expressed explicitly using user-defined constraints. Before making use of this kind of information, however, we must first transform these abstract representations into more concrete code using primitive functions. Since `where` and `assert` constraints are comma-separated lists of expressions, we can distinguish each sub-constraint expression according to comma, for example, the function in Fig. 10 has an `assert` expression that contains three constraints.

It is convenient to represent user-defined constraints by primitive functions in SAC. At compile time, compiler does code transformation to represent user-defined constraints by primitive function. In the following, the detail of transformation will be given. Some of `where` and `assert` expressions can be transformed into builtin functions directly. However, some user-defined constraints may introduce new implicit constraints. For example, if programmers use `shape(A)[2]` to query for the third axis of array `A` while `A` actually is a matrix (i.e. 2-dimensional), the index is out of bounds. Therefore, the compiler must represent these implicitly induced constraints as further explicit constraints to ensure user-defined constraints are valid.

In the following, we will take `where` expression as example to show the representation of constraints.

5.1 Equality of dimensionality

Even though in previous function definition, we did not introduce the constraint on rank of array, it may be used in some applications. Lets assume there is a constraint on rank of parameters A and B in a generic function, i.e. `dim(A)==dim(B)`.

Here, `==` is a user-defined overloaded function. SAC compiler resolve overloading of operations into vectors or scalar operations depending on type of arguments. The scalar operation (`eq_SxS`) just evaluates the equality of scalar values, while the vector operation (`eq_VxV`) introduces an implicit constraint, i.e., it only compares elements of vectors without checking the size of vectors, because this tacitly assumes their lengths coincide.

Since the return value of dim(A) is scalar, the constraint can be represented by primitive function `eq_SxS`.

where(dim(A)==dim(B)) $\Longrightarrow$ where(eq_SxS(dim(A),dim(B)))

5.2 Equality of dedicated axis

The constraint (`shape(A)[i]==shape(B)[j]`) can be transformed into the equality of two scalar values as well. To make the example more general, integers i, j, m, n etc. are used to indicate the index of shape of parameters in the user-defined constraints. However, be aware of an implicit restriction underlying the constraint, i.e. the index of `shape(A)` and `shape(B)` have to be valid. The following transformation makes the implicit constraints explicitly.

```
where(shape(A)[i]==shape(B)[j]) ==>
  where(gt_SxS(dim(A),i);
        gt_SxS(dim(B),j);
        eq_SxS(shape(A)[i],shape(B)[j]))
```

`gt_SxS` is a builtin function which evaluate to true if its first argument is greater than the second one.

5.3 Equality of shape

Abstract user-defined constraints are more powerful at representing equality of dedicated axis. However, because of its abstract and generic there are some implicit constraints underlying. To evaluate equality of shape, we should compare the rank of two parameters first, if that hold, we evaluate their shape equality. The primitive function `eq_VxV` evaluates whether two vectors are equal or not.

```
where(shape(A)==shape(B)) ==>
  where(eq_SxS(dim(A),dim(B)),
        eq_VxV(shape(A),shape(B)))
```

5.4 Arbitrary equality of shape axes

Arbitrary equality of shape axes is much more powerful and complicated. The complexity and further implicit constraints make transformation much more intricate. Implicit constraints underlying in primitive function `tile` is shown as following. Here, `ge_SxS` is a builtin function which evaluate to true if its first argument is greater than or equal to the second one.

```
where(tile([i],[j],shape(A))
        ==tile([m],[n],shape(B))) ==>
  where(eq_SxS(i,m),
        ge_SxS(dim(A),i+j),
        ge_SxS(dim(B),m+n),
        eq_VxV(tile([i],[j],shape(A)),
               tile([m],[n],shape(B))))
```

6. Wrap user-defined constraints into program

At some point, user-defined constraints must be inserted into code. There are several points should be considered to make constraints more useful when assert. First, only if `where` constraints evaluate to true, callee function will be executed, otherwise the program should terminate with an error. Second, if `assert` constraints do not hold, program should terminate with an error message. Third, user-defined constraints can be resolved as far as possible. Fourth, constraints should be accessible to compiler optimization, and knowledge gained by evaluating constraints should be propagated as far as possible.

6.1 Where to insert constraints into code

There are several possible strategies to insert and thus evaluate `where` and `assert` constraints. They can be inserted into caller function or callee function or even both.

Precondition Intuitively, it is feasible to insert the where expression into the caller function before applying arguments to callee function. Reasons to do this are as follows:

1. The callee function will not be executed if `where` constraints do not hold, and the program will be terminated.
2. More importantly, putting constraints into the caller function is crucial for the compiler to resolve constraints statically because shape information on arguments can only be acquired in the caller function.

However, we have to consider another aspect: can these constraints be used in callee function? No matter whether `where` expression evaluates to true or not, for callee function, these constraints will be treated as static knowledge, since once callee function is evaluated, it means the precondition holds. We need to insert these preconditions into the callee function as statically known knowledge, which can benefit further optimizations. Therefore, where constraints should be inserted into both caller and callee functions.

Postcondition The `assert` expressions include constraints among return values and constraints between return values and parameters. For `assert` constraints among return values, it is better to insert constraints into callee function since shape information of return values could be only acquired inside the callee function body, which is a prerequisite for the compiler to resolve any constraint statically. For `assert` constraints between arguments and parameters, it seems a bit complicated, because information of return values and parameters can be acquired from callee and caller function, respectively.

In fact, the situation regarding postconditions is quite similar to that of preconditions. Whenever program execution returns to a caller function, this sheer fact means that the postcondition holds. Thus, we can use the postcondition as static knowledge in the caller function subsequent to the function application itself..

Therefore, we use redundant insertion methods to insert both `where` and `assert` constraints into both callee function and caller function.

6.2 How to insert constraints into code

Insert by conditionals The intuitive way to insert constraints is wrapping user-defined constraints and their implicit constraints into a conditional statement. Using conditional statement, callee function will be evaluated only if parameters satisfy constraints. Since the `where` expressions are inserted into caller function, it is possible to use implicit knowledge of parameters in caller function to resolve constraints as well. However, with respect to fourth point mentioned in the beginning of this section, this method is not optimal, because the result of constraints is only available within the scope of conditional. Optimization or partial evaluation cannot exploit these additional knowledges. Therefore, this kind of insertion is not good enough.

```
1  float[.,.] bar(int x, int y, int z, int w)
2  {
3    ...
4    A = with{([0,0]<=[iv]<[x,y]):1f;}:
5              genarray([x,y]);
6    B = with{([0,0]<=[iv]<[z,w]):2f;}:
7              genarray([z,w]);
8    C = matmul(A,B);
9    ...
10   return C;
11 }
```

Figure 13. Function definition of bar which call function matmul

```
1  float[.,.] bar(int x, int y, int z, int w)
2  {
3    ...
4    A = with{([0,0]<=[iv]<[x,y]):1f;}
5          :genarray([x,y]);
6    B = with{([0,0]<=[iv]<[z,w]):2f;}
7          :genarray([z,w]);
8
9    g1 = guard(gt_SxS(dim(A),0));
10   g2 = guard(gt_SxS(dim(B),1));
11   g3 = guard(eq_SxS(shape(A)[1],shape(B)[0]));
12   A = after_guard(A,g1,g2,g3);
13
14   C = matmul(A,B);
15
16   g4 = guard_hold(gt_SxS(dim(A),0);
17   g5 = guard_hold(gt_SxS(dim(B),1);
18   g6 = guard_hold(gt_SxS(dim(C),1);
19   g7 = guard_hold(
20             eq_SxS(shape(C)[0],shape(A)[0]));
21   g8 = guard_hold(
22             eq_SxS(shape(C)[1],shape(B)[1]));
23   C1 = after_guard(C,g4,g5,g6,g7,g8);
24   ...
25   return C1;
26 }
```

Figure 14. Function bar after code transformation

Using explicit evidence To avoid that problem, we use explicit evidence[3] to insert constraints and weave these contracts into the dataflow. We implement this by using a primitive function `guard` [4], which return true if its argument evaluates to true, otherwise it terminates further evaluation and reports the error.

As discussed above, the resolved constraints have to become a property of program that can be used in the further evaluation. Therefore, we introduce a new kind of guard function named `guard_hold`. It always evaluates to true and mainly serves to provide additional knowledge for further evaluation. Before code generation `guard_hold` annotations will uniformly be removed.

Let's assume there is a piece of code in Fig. 13 that call function matmul, as defined in Fig. 11. The caller function in Fig. 14 contains addtional guard function, i.e. `g1`, `g2` are constraints introduced by `g3`. In Fig. 15, `g1`, `g2`, `g3` become a static knowledge for callee function, which is used for further evaluation. We use `guard_hold` to indicate its argument evaluates to true. `after_guard` takes two or more arguments, and the result of it is the first argument if all consecutive arguments evaluate to true; otherwise, it terminate evaluation.

```
1  float[.,.] x matmul(float[.,.] A,float[.,.] B)
2  {
3    g1 = guard_hold(gt_SxS(dim(A),0));
4    g2 = guard_hold(gt_SxS(dim(B),1));
5    g3 = guard_hold(
6           eq_SxS(shape(A)[1],shape(B)[0]));
7    A  = after_guard(A,g1,g2,g3);
8
9    BT = transpose(B);
10   a0 = shape(A)[0];
11   b1 = shape(B)[1];
12   C = with {
13     ([0,0]<=[i,j]<[a0,b1]):sum(A[i] * BT[j]);
14     }:genarray([a0,b1],0f);
15
16   g6 = guard(gt_SxS(dim(A),0));
17   g7 = guard(gt_SxS(dim(B),1));
18   g8 = guard(gt_SxS(dim(C),1));
19   g9 = guard(eq_SxS(shape(C)[0],shape(A)[0]));
20   g10 = guard(
21           eq_SxS(shape(C)[1],shape(B)[1]));
22   C1 = after_guard(C,g6,g7,g8,g10);
23
24   return C1;
25 }
```

Figure 15. Function matmul after code transformation

If the constraints can be resolved at compile time, the guard function may be removed partially by partial evaluation. Here, `guard_hold` will stay and conditional removed after partial evaluation.

7. Syntactic sugar for shape constraints

Our shape constraints are a syntactically restricted form of Boolean expressions. While this approach provides a certain degree of generality, phrasing of shape constraints can often be significantly simplified by adding some syntactic sugar to the specification of types.

The matrix multiplication example discussed throughout the paper illustrates the common need to access extents of argument arrays along individual axes. In Fig. 6 we used the term `shape(A)[1]` to express this. Fig. 16 shows the same example with a little bit of syntactic sugar: by using identifiers instead of dots in AKD types, we can elegantly bind identifiers to the extents of argument arrays along relevant axes.

```
1  float[.,.] matmul(float[.,a1] A,float[b0,.] B)
2  where (a1==b0)
3  {
4    /*Computation*/
5  }
```

Figure 16. Matrix multiplication of Fig. 6 with syntactic for sugar constraints

We can even go one step further and express equality constraints by repeatedly using the same identifier instead of anonymous dots. Fig. 17 demonstrates this with a complete description of the inherent shape constraints of matrix multiplication, without making any use of any explicit `where` and `assert` constraints.

```
1  float[x,z] matmul(float[x,y] A,float[y,z] B)
2  {
3    /*Computation*/
4  }
```

Figure 17. Fully sugared expression of matrix multiplication shape constraints

Desugaring of the above examples into explicit constraints is a fairly straightforward preprocessing step.

8. Related work

Programming errors are common in software system and hard to detect. Much research attention has been paid to error detection. One popular approach is *design by contract* proposed by Bertrand Meyer [5, 6], which is widely used both in object-oriented programming languages [7, 8] and functional programming languages [3, 9, 10]. But none of them has concrete shape restrictions on arrays as described in this paper.

Interpreted array programming languages like APL[11], J [12] are dynamically typed. When the interpreter encounters an array operation, it checks whether its arguments are proper, if so, performs computation by invoking native implementation, otherwise, aborts program with error message. Without a priori static analysis makes bugs hard to find, and What's more, this design decision has a considerable runtime impact [13]. Our proposed compiler technology can counter these issues. The SAC compiler tries to statically resolve all user-defined constraints. If the constraints do not hold, it reports error message. For the constraints that can not be resolved at compile time, we will leave them to dynamic check.

Our work indeed is similar to Qube [14, 15], a programming language that combines the expressiveness of array programming with the power of dependent type. We try to express implicit constraints explicitly and resolve constraints statically. However, our approach is much more programmer friendly and expressiveness, and we eliminate the effort of learning new logic with its theorem prover.

Another related work was carried out by Herhut et al.[4], which focuses on checking domain constraints for built-in functions. In contrast our approach targets general user-defined functions and thus more general constraints.

To resolve these constraints, we may adopt *symbiotic expressions* [16] which is a method for algebraic simplification within a compiler. Even though currently *symbiotic expression* in SAC is compiler-generated expression, we may be able to reuse some technology to support our user-defined constraints.

9. Conclusion

This paper proposes a compiler technology called user-defined constraints that allows programmers to express shape constraints on parameters (`where`) and return values (`assert`) explicitly in function level. It help compiler to generate more reliable executable code by restrict function definition, improve performance through better optimization and provide a means for software documentation, which helps programmers to better understand code. Using this approach, programmers neither need to add redundant conditional code in caller function, nor add constraints in callee function. User-defined constraints will be transformed into primitive functions in SAC, and then inserted into program properly according to their type.

We mainly focus our presentation on introducing user-defined constraints and its implicit constraints and code transformation in this paper. It remains as future research to investigate how compiler resolves more constraints statically and effectively, and whether this technology brings the properties of our type system close to strongly typed system based on dependent types.

References

[1] Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427

[2] Grelck, C.: Single assignment c (sac): High productivity meets high performance. In Z. Horvath, V.Z., ed.: 4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary. Volume 7241 of Lecture Notes in Computer Science., Springer (2012) to appear.

[3] Xu, D.N.: Extended static checking for Haskell. In: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell. Haskell '06, New York, NY, USA, ACM (2006) 48–59

[4] Herhut, S., Scholz, S.B., Bernecky, R., Grelck, C., Trojahner, K.: From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In Chitil, O., Horváth, Z., Zsók, V., eds.: 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., Springer (2008) 254–273

[5] Meyer, B.: Eiffel: The language and environment. Prentice Hall Press, 300 (1991)

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

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

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

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

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

[11] Iverson, K.: A programming language. In: Proceedings of the May 1-3, 1962, spring joint computer conference, ACM (1962) 345–351

[12] Iverson, K.: J Introduction and Dictionary, New York, NY, USA. (1995)

[13] Bernecky, R.: Reducing computational complexity with array predicates. In: ACM SIGAPL APL Quote Quad, ACM (1998) 39–43

[14] Trojahner, K.: QUBE–Array Programming with Dependent Types. PhD thesis, University of Lübeck, Lübeck, Germany (2011)

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

[16] Bernecky, R., Herhut, S., Scholz, S.: Symbiotic expressions. Implementation and Application of Functional Languages (2011) 107–124