Single Assignment C (SAC)
High Productivity Meets High Performance

Clemens Grelck

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

Abstract. We present the ins and outs of the purely functional, data parallel programming language SaC (Single Assignment C). SaC defines state- and side-effect-free semantics on top of a syntax resembling that of imperative languages like C/C++/C# or Java: functional programming with curly brackets. In contrast to other functional languages data aggregation in SaC is not based on lists and trees, but puts stateless arrays into the focus.

SaC implements an abstract calculus of truly multidimensional arrays that is adopted from interpreted array languages like Apl. Arrays are abstract values with certain structural properties. They are treated in a holistic way, not as loose collections of data cells or indexed memory address ranges. Programs can and should be written in a mostly index-free style. Functions consume array values as arguments and produce array values as results. The array type system of SaC allows such functions to abstract not only from the size of vectors or matrices but likewise from the number of array dimensions, supporting a highly generic programming style.

The design of SaC aims at reconciling high productivity in software engineering of compute-intensive applications with high performance in program execution on modern multi- and many-core computing systems. While SaC competes with other functional and declarative languages on the productivity aspect, it competes with hand-parallelised C and Fortran code on the performance aspect. We achieve our goal through stringent co-design of programming language and compilation technology.

The focus on arrays in general and the abstract view of arrays in particular combined with a functional state-free semantics are key ingredients in the design of SaC. In conjunction they allow for far-reaching program transformations and fully compiler-directed parallelisation. From literally the same source code SaC currently supports symmetric multi-socket, multi-core, hyperthreaded server systems, CUDA-enables graphics accelerators and the MicroGrid, an innovative general-purpose many-core architecture.

The CEFP lecture provides an introduction into the language design of SaC, followed by an illustration of how these concepts can be harnessed to write highly abstract, reusable and elegant code. We conclude with outlining the major compiler technologies for achieving runtime performance levels that are competitive with low-level machine-oriented programming environments.

1 Introduction and Motivation

The on-going multi-core/many-core revolution in processor architecture has arguably more radically changed the world's view on computing than any other innovation in microprocessor architecture. For several decades the same program could be expected to run faster on the next generation of computers than on the previous. The trick that worked so well and so cheaply all the time is clock frequency scaling. Gordon Moore's famous prediction (also known as Moore's law) says that the number of transistors in a chip doubles every 12–24 months [1]. In other words, the number of transistors on a single chip was predicted to grow exponentially. Surprisingly, this prediction has been fairly accurate since the 1960s. Beyond all clever tricks in microprocessor architecture that were enabled by ever growing transistor counts the probably most important impact of Moore's law lies in the miniaturisation of the logical structures within a processor. The time it takes for an electrical signal to advance from one gate to the next is linear in the distance. With the distance shrinking exponentially, processors were able to run on higher and higher clock frequencies, moving from kilo-Hertz to giga-Hertz.

But now this "free lunch" of programs automatically running faster on a new machine is over [2]. What has happened? Unlike Moore's law, which is rather a prediction than a law, there are also true laws of physics, and according to them the energy consumption of a processor grows quadratically with the clock frequency. Consequently, energy cost has become a relevant factor in computing these days. Another law of physics, the law of conservation of energy, says that energy neither appears from nothing nor does it disappear to nothing; energy only changes its physical condition. In the case of processors, the electrical energy consumed is mostly dissipated as heat, thus requiring even more energy for cooling. These cause the technical and economic challenges we face today.

Circumstances have fostered two technological developments: the multi-core revolution and the many-core revolution. The former means that general-purpose processors do not run at any higher clock frequency than before, but the continuing miniaturisation of structures is used to put multiple cores, fully-fledged processors by themselves, into a single chip. While quad-core processors are already common place in the consumer market, server processors often have already 6, 8 or even 12 cores today. It is generally anticipated that Moore's law of exponential growth will continue for the foreseeable future, but that instead of the clock frequency the number of cores will benefit.

The many-core revolution has its origin in a similar technological progress in the area of graphics cards. With their specialised designs graphics cards have developed into highly parallel, extremely powerful co-processors. They can compute fitting workloads much faster than state-of-the-art general-purpose processors. And, increasingly relevant, they can do this with a fraction of the energy budget. With the fairly general-purpose CUDA programming model, particularly NVidia graphics cards have become integral parts of many high-performance computing installations [3]. But even on the other end of the scale, in the personal computing domain, GPGPUs (or general-purpose graphics processing units) have become relevant for computing beyond computer graphics. After all, every computer does have a graphics card, and its full potential is not always needed for merely controlling the display.

Looking into the future (which is always dangerous) one can anticipate a certain symbiosis of general-purpose multi-core processors and GPU-style accelerators into unified processor designs with a few general-purpose *fat* cores and a large number of restricted *thin* cores. AMD's Fusion and Intel's Knights Ferry architectures are precursors of this development.

The radical paradigm shift in computer architecture from increasing clock frequencies to duplicating computing devices on chip incurs a paradigm shift in software engineering that is at least as revolutionary. As said before, programs no longer automatically benefit from a new generation of computers. A sequential program does not run any faster on a quad-core system than on a uni-core system, and it is very unlikely that it takes advantage of a computer's graphics card. Software at any level must be parallelised to effectively take advantage of today's computers. Harnessing the full power of increasingly concurrent, increasingly diverse and increasingly heterogeneous chip architectures is a challenge for future software engineers.

The multicore revolution must have a profound impact on the practice of software engineering. While parallel programming per sé is hardly new, until very recently it was largely confined to the supercomputing niche. Consequently, programming methodologies and tools for parallel programming are geared towards the needs of this domain: squeezing the maximum possible performance out of an extremely expensive computing machinery through low-level machine-specific programming. Programming productivity concerns are widely ignored as running code is often more expensive than writing it.

What has changed with the multi-/many-core revolution is that any kind of software and likewise any programmer is affected, not only specialists in high performance computing centers with a PhD in computer science.

What has also changed thoroughly is the variety of hardware. Until recently, the von-Neumann model of sequential computing was all that most software engineers would need to know about computer architecture. Today's computer landscape is much more varied and with existing programming technology this variety immediately affects programming. A computer today may just have a single dual-core or quad-core processor, but it may likewise be a 4-processor system with 4, 6 or 12 cores per processor [4,5]. So, already today the number of cores in a general-purpose system can differ by more than one order of magnitude. Technologies such as Intel's hyperthreading [6] further complicate the situation: they are often presented as real cores by the operating system, yet they require a different treatment.

Non-x86 based processor architectures like Oracle's Niagara range offer even more parallelism. The T3-4 server system [7,8] shipped in 2011, for instance, features 4 processors with 16 cores each while each core supports 8 hardware threads. Such a system totals in 512 hardware threads and adds another order of magnitude to the level of parallelism that software needs to effectively take advantage of. A similar variety of technologies can be seen in the GPGPU market.

Now any multi-core system can freely be combined with one or even multiple GPGPU accelerators leading to a combinatorial explosion of possibilities. This, at the latest, makes it technologically and economically challenging to write software that makes decent use of a large variety of computing systems.

The quintessential goal of the SaC project lies in the co-design of programming language technology and the corresponding compiler technology that effectively and efficiently maps programs to a large variety of parallel computing architectures [9,10]. In other words, SaC aims at reconciling programming productivity with execution performance in the multi-/many-core era.

Our fundamental approach is *abstraction*. In analogy to the von Neumann architecture of sequential computing machines SaC abstracts from all concrete properties of computing systems and merely allows the specification of concurrent activities without any programmer control as to whether two concurrent activities are actually evaluated in parallel or sequentially. This decision is entirely left to the compiler and runtime system. The guiding principle is to let the programmer define *what* to compute, not *how* exactly this is done. Our goal is to put expert knowledge, for instance on parallel processing or computer architecture, once into compiler and runtime system and not repeatedly into low-level implementations of many application programs. This approach is particularly geared towards the overwhelming number of software engineers who are neither experts in parallel programming nor appreciate being forced to develop such skills. Nonetheless, it is particularly this target group that wants or must exploit the capabilities of modern multi-core and many-core computing systems with limited software engineering effort.

Specifying *what* to compute, not exactly *how* to compute sounds very familiar to functional programmers. And indeed, SaC is a purely functional language with a state- and side-effect-free semantics. Thus, SaC programs deal with values, and program execution computes new values from existing values in a sequence of context-free substitution steps. How values actually manifest in memory, how long they remain in memory and whether they are created at all is left to the language implementation. Abstracting from all these low-level concerns makes SaC programs expose the algorithmic aspects of some computation because they are not interspersed with organisational aspects of program execution on some concrete architecture.

In order to make exclusively compiler-directed parallelisation feasible, SaC embraces a data-parallel agenda. More precisely, SaC is an *array programming language* in the tradition of Apl [11,12], J [13] or Nial [14]. In fact, multi-dimensional arrays are the basic data aggregation principle in SaC. Operations on arrays are defined not exclusively but overwhelmingly following a data-parallel approach. Before we look closer into SaC, let us first illustrate why the data-parallel approach is crucial for our goal of supporting a wide range of parallel architectures solely through compilation. We do this by means of an example algorithm that clearly is none of the usual suspects in (data-)parallel computing. Fig. 1 shows three different implementations of the factorial function: an imperative implementation using C, a functional implementation in OCaml and a data-parallel implementation in SaC. It is characteristic for both the imperative and the functional definition of the factorial function that they do not expose any form of concurrency suitable for compiler-directed parallelisation. The imperative code is sequential by definition, but its functional counterpart likewise leads to a purely sequential computation.

```
int fac( int n)
{
  f = 1;
  while (n > 1) {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

```
fac n = if n <= 1
        then 1
        else n * fac (n - 1)
```

```
int fac( int n)
{
  return prod( 1 + iota( n));
}
```

Fig. 1. Three definitions of the factorial function: imperative using C (top), functional using OCaml (middle) and data-parallel using SaC (bottom)

In contrast, the array-style SaC implementation of the factorial function does expose a wealth of concurrency to compiler and runtime system to exploit for automatic parallelisation. However, this admittedly warrants some explanation of the SaC code in Fig. 1. The `iota` function (the name is inspired by the corresponding Apl operation) yields a vector (a one-dimensional array) of `n` elements with the values `0` to `n-1`. Adding the value `1` to this vector yields the `n`-element vector with the numbers `1` to `n`. Computing the product of all elements of this vectors yields the factorial of `n`. While this definition of the factorial function may be unusual at first glance, it offers one significant advantage over the other definitions of Fig. 1: it exposes concurrency. Each of the three conceptual steps is a data-parallel operation. For appropriate values of `n` the data-parallel formulation of the factorial function exposes a high degree of fine-grained concurrency.

Fig. 2 illustrates why this is highly relevant. In the example we compute the factorial of 10. The data-parallel specification based on `iota`, element-wise addition and `prod` exposes a 10-fold concurrency in computing the factorial number. This, however, does not mean that the computation is split into 10 independent tasks, processes or threads. It is merely an option for the compiler and runtime system to exploit this fine-grained concurrency. Depending on the target

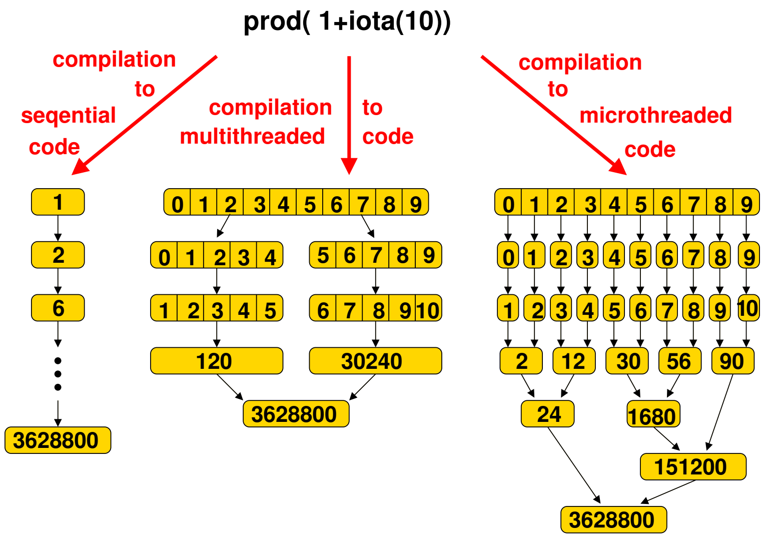


Fig. 2. Design choices in compiling a data-parallel program

architecture this may or may not be the case. If the target architecture, for instance, supports very light-weight concurrent activities, the compiler may indeed decide to expose the full amount of concurrency to the hardware. The MicroGrid many-core architecture [15] is such an example.

On the other end of the spectrum compiler and runtime system may equally well decide to run the entire computation sequentially. Maybe we utilise sequential legacy hardware; maybe we have already exhausted our parallel computing capabilities on an outer application level. If so, it is fairly simple *not* to make use of the apparent concurrency and generate sequential binary code instead.

Between these two extremes, exploiting all concurrency available or exploiting no concurrency at all, we find an almost contiguous design space for compiler and runtime system to make appropriate decisions. The right choice depends on a variety of concerns including properties of the target architecture, characteristics of the code and attributes of the data being processed. For example, in the center of Fig. 2 we can identify a solution that is presumably well-suited for a dual-core system. The compiler generates two independent tasks that each take care of one half of the intermediate vector. Both threads can run without synchronisation until the final multiplication of the partial reduction results.

A compiler could even generate multiple alternative code versions and postpone any decision until runtime when complete information about hardware capacities, data sizes, etc, are available to make a much more educate choice.

Of course, the factorial function is merely an example to illustrate the principle of data-parallel array programming, not at all a relevant application. Neither is the factorial function particularly interesting to be computed in parallel for large argument numbers, nor do the concrete implementations of Fig. 1, based on machine-width integer representations, support sufficiently large values.

As the name Single Assignment C suggests and the factorial example already reveals to some extent, SAC does not follow regular syntactic conventions of established functional languages. Neither do we invent a completely new syntax from scratch. Instead, we aim at providing imperative programmers with the warm feeling of a familiar programming environment. After all, the majority of programmers suddenly confronted with the multi-core revolution has not used HASKELL, OCAML or CLEAN before but rather C, C++, C# or JAVA. We will later see how imperative appearance and functional semantics can make a very beneficial symbiosis.

Last but certainly not least, SAC aims at combining high-level, problem-oriented programming not only with fully automatic parallelisation but likewise with competitive sequential performance. And competition here means established imperative programming languages, not high-level, declarative or functional ones. If we aim at converting imperative programmers to SAC, we must be able to generate absolute performance gains through automatic parallelisation. In other words SAC aims at outperforming sequential imperative codes on parallel hardware. For that it is paramount to deliver sequential performance that is close to imperative programs. After all, we cannot expect more than a linear performance increase from parallelisation. To support the performance demands, SAC dispenses with a number of programming features typical for main-stream, general-purpose functional languages. For instance, SAC neither supports higher-order functions, nor currying or partial applications. SAC also follows a strict evaluation regime.

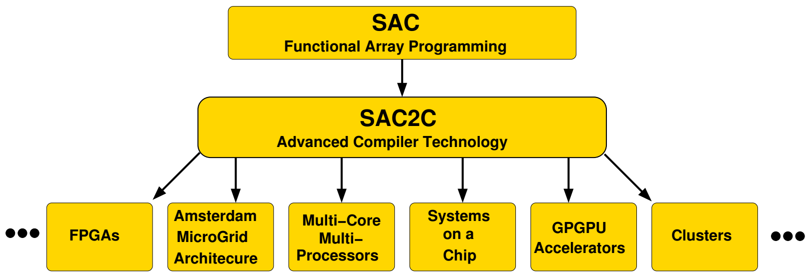

SAC
Functional Array Programming
SAC2C
Advanced Compiler Technology
FPGAs
Amsterdam MicroGrid Architecure
Multi–Core Multi–Processors
Systems on a Chip
GPGPU Accelerators
Clusters

Fig. 3. The SAC compilation challenge: past, present and future work

Fig. 3 illustrates the compilation challenge taken by SAC. Based on competitive sequential performance, we aim at compiling a single SAC source program to a variety of computing architectures. At the moment SAC supports symmetric (potentially hyper-threaded) multi-core multi-processor systems with shared memory, i.e. today's bread-and-butter server systems. Moreover, SAC also supports general-purpose graphics processing units (GPGPUs) as accelerators as well as the MicroGrid [15], an innovative general-purpose many-core processor architecture developed at the University of Amsterdam. Work is currently ongoing to combine multi-core and many-core code generators to support hybrid systems-on-chip. Support for reconfigurable hardware on one end of the spectrum and network-interconnected clusters of multi-core servers with accelerators on the other mark up-coming challenges that we have only started exploring.

The rest of the article is organised as follows: We begin with the core language design of SaC and explain the relationship between imperative syntax and functional semantics in Section 2. Section 3 elaborates on the calculus of multidimensional arrays and discusses its implementation by SaC. We then introduce the array type system of SaC and the associated programming methodology in Section 4. Sections 5 and 6 illustrate programming in SaC by means of two case studies: variations of convolution and numerical differentiation. Sections 7, 8 and 9 complete the introductory text on SaC and explain the module system, SaC's approach to functionally sound input/output and the foreign language interfaces, respectively. Last not least, we discuss essential aspects of the SaC compiler and runtime system in Section 10. A small selection of related work is sketched out in Section 11 before we conclude with a short summary and outlook on current and future research directions in Section 12.

2 Core Language Design

In this section we describe the core language design of SaC. First, we identify the syntactical subset of C for which we can define a functional semantics as language kernel for SaC (Section 2.1). Afterwards, we explain the relationship between the imperative, C-inspired syntax and its truly functional semantics in detail (Section 2.2).

2.1 A Functional Subset of ISO C

The core of SaC is the subset of ANSI/ISO C [16] for which functional semantics can be defined (surprisingly straightforwardly). Fig. 4 illustrates the similarities and differences between SaC and C. In essence, SaC adopts from C the names of the built-in types, i.e. `int` for integer numbers, `char` for ASCI characters, `float` for single precision and `double` for double precision floating point numbers. Conceptually, SaC also supports all variants derived by type specifiers such as `short`, `long` or `unsigned`, but for the time being we merely implements the above standard types. Unlike C, SaC properly distinguishes between numerical, character and Boolean values and features a built-in type `bool` for the latter.

As a functional language SaC uses type inference instead of C-style type declarations. This requires a strict separation of values of different basic types. While type `bool` is, as expected, inferred for the Boolean constants `true` and `false` and character constants like `'a'` are obviously of type `char`, the situation is less clear for numerical constants. Here, we decide that any number constant without decimal point or exponent specification is of type `int`. Any floating

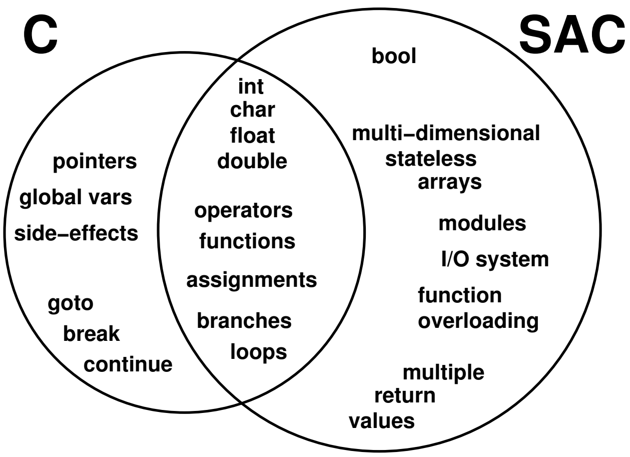


Fig. 4. Similarities and differences between SAC and C

point constant with decimal point or exponent specification is by default of type `double`. A trailing `f` character makes any numerical constant a single precision floating point constant, and a trailing `d` character a double precision floating point constant. For example, `42` is of type `int`, `42.0` is of type `double`, `42.0f` and `42f` are of type `float` and `42d` is again of type `double`. SAC requires explicit conversion between values of different basic types by means of the overloaded conversion functions `toi` (conversion to integer), `toc` conversion to character, `tof` (conversion to single precision floating point), `tod` (conversion to double precision floating point) and `tob` (conversion to Boolean).

Despite these minor differences in details, SAC programs generally look intriguingly similar to C programs. SAC adopts the C syntax for function definitions and function applications that clearly distinguishes between functions and values. Function bodies are essentially sequences of assignments of expressions to variables. While C-style variable declarations are superfluous due to type inference, they are nonetheless permitted and may serve documentation purposes. If present declared types are checked against inferred types.

In addition to constants as explained above, expressions are made up of identifiers, function applications and operator applications. SAC supports most operators from C, among them all arithmetic, relational and logical operators. As usual, Boolean conjunction and disjunction only evaluate their right operand expression if necessary. Furthermore, SAC does also support the tertiary conditional expression operator from C (question mark and colon, in other words a proper functional conditional), operator assignments (e.g. `+=` and `*=`) as well as pre and post increment and decrement operators (i.e. `++` and `--`). For the time being, SAC does not support the bitwise operations of C.

SAC adopts almost all of C's structured control flow constructs: branches with and without alternative (`else`), loops with leading (`while`) and with trailing (`do...while`) predicate and, last not least, counted loops (`for`). All of these constructs feature exactly the same syntax as C proper. In case of the `for`-loop we even adopt the definition of exact semantics as a (preprocessor) transformation into a `while`-loop [17]. Given the proper separation between Boolean and numerical values, predicates in branches, conditional expressions and loops must be expressions of type `bool`, not `int` as in C.

We do not mention C's `switch`-construct in Fig. 4. While the SaC compiler does not implement this for the time being, our choice is not motivated by any conceptual issues, but solely by engineering effort concerns. In contrast, C does have a number of quintessentially imperative features that we definitely do not want to adopt: pointers, global variables and side effects in general. Moreover, C-style control flow manipulation features, such as `goto`, `break` and `continue`, make no sense in SaC because the functional semantics dispenses with any form of control flow.

```
int gcd( int high, int low)
{
  if (high < low) {
    mem  = low;
    low  = high;
    high = mem;
  }

  while (low != 0) {
    quotient, remainder = diffmod( high, low);
    high = low;
    low  = remainder;
  }

  return high;
}

int, int diffmod( int x, int y)
{
  quot   = x / y;
  remain = x % y;
  return (quot, remain);
}

int main()
{
  return gcd( 22, 27);
}
```

Fig. 5. Example of a core SaC program that illustrates the similarities and differences between SaC and C: greatest common denominator following Euclid's algorithm

The language kernel of SaC is enriched by a number of features as illustrated in Fig. 4. Some of these features are characteristic for SaC, e.g. the multi-dimensional, stateless arrays. Others are mere programming conveniences or state-of-the-art modernisations of C, e.g. a proper module system with information hiding or an I/O system that combines the simplicity of imperative I/O (e.g. simply adding a print statement where one is needed) with a save integration of state manipulation into the purely functional context of SaC "under the hood". Unlike C but in the tradition of C++ SaC also supports function and operator overloading (ad-hoc polymorphism). Syntactically, SaC allows functions to instantaneously yield multiple values. As functions can (of course) take multiple arguments, support for multiple return values creates a nice symmetry between domain and codomain.

Fig. 5 illustrates the (scalar) language kernel of SaC by means of a simple example: Euclid's algorithm to determine the greatest common denominator of two natural numbers. The code in Fig. 5 mainly highlights the syntactical similarity (if not identity) between SaC and C (at least for such simple programs). The code, nonetheless, is not legal C code as it also showcases a SaC-specific language feature: functions with multiple return values. The auxiliary function `diffmod` instantaneously yields the quotient and the remainder of two integers. Consequently, the function `diffmod` is defined to yield two integer values and its `return`-statement contains two expressions. Parentheses are required around multiple return expressions. The application of `diffmod` demonstrates instantaneous variable binding. Like in C and other languages, a function with the reserved name `main` defines the starting point of program execution. One may note the complete absence of local variable declarations in Fig. 5.

2.2 Functional Semantics vs C-Like Syntax

Despite its imperative appearance, SaC is a purely functional programming language. While we refrain from any attempt to define a formal functional semantics for the language kernel, we nonetheless illustrate the main ideas behind combining an imperative syntax with a purely functional semantics. The examples in Figs. 6, 7 and 8 show relevant fragments of SaC code and explain their exact meaning by semantically equivalent OCaml code.

```
int add1( int a, int b)
{
  c = a + b;
  x = 1;
  c = c + x;
  return c;
}
```

$\Longleftrightarrow$

```
let add1 (a,b) =
  let c = a + b
  in let x = 1
     in let c = c + x
        in c
```

Fig. 6. Semantic equivalence between SaC and OCaml: simple function definitions

Fig. 6 shows a very simple SAC function `add1` whose body merely consists of a sequence of assignments of expressions to variables and a trailing `return`-statement. Semantically, we interpret a sequence of assignments as a sequence of nested `let`-expressions with the return expression serving as the final goal expression of the `let`-cascade. This transformational semantics easily clarifies why and how SAC, despite prominently featuring the term *single assignment* in its name, does actually allow repeated assignment to the "same" variable. Any assignment to a previously defined variable or function parameter is actually an assignment to a fresh variable that merely happens to bear the same name as the variable defined earlier. Standard scoping and visibility rules, even familiar to imperative programmers, clarify that the previously assigned variable can no longer be accessed.

```
int fac( int n)
{
  if (n>1) {
    r = fac( n-1);
    f = n * r;
  }
  else {
    f = 1;
  }
  return f;
}
```

$\Longleftrightarrow$

```
let fac n =
  if n>1
  then let r = fac (n-1)
       in let f = n * r
          in f
  else let f = 1
       in f
```

Fig. 7. Semantic equivalence between SAC and OCAML: branching

The functional interpretation of imperative branching constructs is shown in Fig. 7 by means of a recursive definition of the factorial function. In essence, we "copy" the common code following the branching construct including the trailing `return`-statement into both branches. By doing so we transform the C branching statement into a functional OCAML conditional expression. For consistency with the equivalence defined in Fig. 6 we also transform both branches into cascading `let`-expressions.

The functional interpretation of loops requires slightly more effort, but it is immediately apparent that imperative loops are mainly syntactic sugar for tail recursion. Fig. 8 demonstrates this analogy by means of a standard imperative definition of the factorial function using a `while`-loop. Here, we need to turn the loop into a tail-recursive auxiliary function (`fwhile`) that is applied to the argument `n` and the start value `f`. Upon termination the auxiliary function yields the factorial.

```
int fac( int n)
{
  f = 1;

  while (n>1) {
    f = f * n;
    n = n - 1;
  }

  return f;
}
```

$\Longleftrightarrow$

```
let fac n =
  let f = 1
  in let fwhile (f,n) =
       if n>1
       then let f = f * n
            in let n = n - 1
               in fwhile (f,n)
       else f
     in let f = fwhile (f,n)
        in f
```

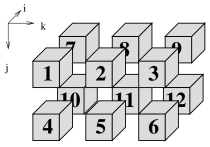
Fig. 8. Semantic equivalence between SaC and OCaml: while-loops

3 Multidimensional Stateless Arrays

On top of the scalar kernel SaC provides genuine support for truly multidimensional stateless arrays. The section begins with introducing the array calculus and its incorporation into a concrete programming language (Section 3.1) and proceeds to the built-in array functions supported by SaC (Section 3.2). The rest of the section is devoted to with-loops, the SaC array comprehension construct. We first introduce the principles (Section 3.3), then we show a complete example (Section 3.4) and finally we provide a complete reference of features (Section 3.5).

3.1 Array Calculus

On top of this language kernel SaC provides genuine support for truly multidimensional arrays. In fact, SaC implements an array calculus that dates back to the programming language Apl[18,11]. This calculus was later adopted by other array languages, e.g. J[19,13,20] or Nial[14,21] and also theoretically investigated under the name ψ-calculus [22,23]. In this array calculus any multidimensional 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 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 along ascending axes with respect to the shape vector, sometimes referred to as *row-major* ordering. Fig. 9 shows a number of example arrays and illustrates the relationships between rank, shape vector and data vector.



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]

Fig. 9. Truly multidimensional arrays in SaC and their representation by data vector, shape vector and rank scalar

More formally, let A be an n-dimensional array represented by the rank scalar n, the shape vector $\boldsymbol{s}_A = [s_0, \ldots, s_{n-1}]$ and the data vector $\boldsymbol{d}_A = [d_0, \ldots, d_{m-1}]$. Then the equation

$$m \quad = \quad \prod_{i=0}^{n-1} \quad s_i$$

describes the correspondence between the shape vector and the length of the data vector. Moreover, the set of legal index vectors of the array A is defined as

$$\mathcal{IV}_A \quad := \quad \{\ [iv_0, \ldots\ , iv_{n-1}] \mid \forall j \in \{0, \ldots\ , n-1\} : 0 \leq\ iv_j < s_j \} \quad .$$

An index vector $\boldsymbol{iv} = [iv_0, \ldots\ , iv_{n-1}]$ denotes the element d_k of the data vector $\boldsymbol{d}_A$ of array A if $\boldsymbol{iv}$ is a legal index vector of A, i.e. $\boldsymbol{iv} \in \mathcal{IV}_A$, and the equation

$$k \quad = \quad \sum_{i=0}^{n-1} \quad (iv_i \ * \prod_{j=i+1}^{n-1} s_j)$$

holds. Two arrays A and B are *conformable* iff they have the same element type and the same number of elements:

$$|\boldsymbol{d}_A| \quad = \quad |\boldsymbol{d}_B|$$

A vector of natural numbers s is *shape-conformable* to an array A iff the product of the elements of the vector equals the number of elements of the array:

$$\prod_{i=0}^{n-1} \quad s_i \quad = \quad |\boldsymbol{d}_A|$$

As already shown in Fig. 9 the array calculus nicely extends to scalars. A scalar value has the rank zero and the empty vector as shape vector; the data vector contains a single element, the scalar value itself. This is completely consistent with the rules sketched out before. The rank determines the number of elements in the shape vector. As the rank of a scalar is zero, so is the number of elements in the shape vector. The product of all elements of the shape vector determines the number of elements in the data vector. The product of an empty sequence of values is one, i.e. the neutral element of multiplication.

Unifying scalars and arrays within a common calculus allows us to say that any value in SaC is an array, and as such it has a rank, a shape vector and a data vector. Furthermore, we achieve a complete separation between data and structural information (i.e. rank and shape).

3.2 Built-In Operations on Arrays

In contrast to all its ancestors, from Apl to the ψ-calculus, SaC only defines a very small number of built-in operations on multidimensional arrays. They are directly related to the underlying calculus:

- `dim(` *a* `)`
  yields the rank scalar of array *a*;
- `shape(` *a* `)`
  yields the shape vector of array *a*;
- `sel(` *iv*`,` *a* `)`
  yields the element of array *a* at index location *iv*, provided that *iv* is a legal index vector into array *a* according to the definition above;
- `reshape(` *sv*`,` *a* `)`
  yields an array that has shape *sv* and the same data vector as array *a*, provided that *sv* and *a* are shape-conformable;
- `modarray(` *a*`,` *iv*`,` *v* `)`
  yields an array with the same rank and shape as array *a*, where all elements are the same as in array *a* except for index location *iv* where the element is set to value *v*.

For the convenience of programmers SaC supports some syntactic sugar to express applications of the `sel` and `modarray` built-in functions:

```
            sel( iv, a)   ≡   a[iv]
a = modarray( a, iv, v);   ≡   a[iv] = v;
```

Fig. 10 further illustrates the SaC array calculus and its built-in functions by a number of examples. Most notably, selection supports any prefix of a legal index vector. The rank of the selected subarray equals the difference between the rank of the argument array and the length of the index vector. Consequently, if the length of the index vector coincides with the rank of the array, the rank of the result is zero, i.e. a single element of the array is selected.

$$
\begin{array}{rcl}
\texttt{vec} & \equiv & \texttt{[4,5,6,7]} \\
\texttt{mat} & \equiv & \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix} \\
\texttt{dim(mat)} & \equiv & \texttt{2} \\
\texttt{shape(mat)} & \equiv & \texttt{[3,4]} \\
\texttt{dim(vec)} & \equiv & \texttt{1} \\
\texttt{shape(vec)} & \equiv & \texttt{[4]} \\
\texttt{mat[[1,2]]} & \equiv & \texttt{6} \\
\texttt{vec[[3]]} & \equiv & \texttt{7} \\
\texttt{mat[[]]} & \equiv & \texttt{mat} \\
\texttt{mat[[1]]} & \equiv & \texttt{vec} \\
\texttt{mat} & \equiv & \texttt{reshape([3,4], [0,1,2,3,4,5,6,7,8,9,10,11])} \\
\texttt{[[4,5],[6,7]]} & \equiv & \texttt{reshape([2,2], vec)}
\end{array}
$$

Fig. 10. SAC built-in functions in the context of the array calculus

3.3 With-Loop Array Comprehension

With only five built-in array operations (i.e. `dim`, `shape`, `sel`, `reshape` and `modarray`) SAC leaves the beaten track of array-oriented programming languages like APL and FORTRAN-90 and their derivatives. Instead of providing dozens if not a hundred or more hard-wired array operations such as element-wise extensions of scalar operators and functions, structural operations like shift and rotate along one or multiple axes and reduction operations with eligible built-in and user-defined operations like sum and product, SAC features a single but versatile array comprehension construct: the WITH-loop.

WITH-loops can be used to implement all the above and many more array operations in SAC itself. We make intensive use of this feature and provide a comprehensive standard library of array operations. Compared to hard-wired array support this approach offers a number of advantages. For instance, we can keep the language design of SAC fairly lean, the library implementations of array operations do not carve their exact semantics in stone and SAC users can easily extend and adapt the array library to their individual needs.

WITH-loops facilitate the specification of `map`- and `reduce`-like aggregate array operations. They come in three variants, named `genarray`, `modarray` and `fold`, as illustrated by means of simple examples in Figs. 11, 12 and 13, respectively. Since the WITH-loop is by far the most important and most extensive syntactical extension of SAC, we also provide a formal definition of the syntax in Fig. 14. For didactic purposes we begin with a simplified form of WITH-loops here and discuss a number of extensions in the following section.

We start with the `genarray`-variant in Fig. 11. Any WITH-loop array comprehension expression begins with the key word `with` (line 1) followed by a *partition* enclosed in curly brackets (line 2), a colon and an *operator* that defines the WITH-loop variant, here the key word `genarray`. The `genarray`-variant is an array comprehension that defines an array whose shape is determined by the first expression following the key word `genarray`. By default all element values

```
A = with {
      ([1,1] <= iv < [4,4]) : e(iv);
    }: genarray( [5,4],  def  );
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]
[3,0]	[3,1]	[3,2]	[3,3]
[4,0]	[4,1]	[4,2]	[4,3]

$\Longrightarrow$

$[\![def]\!]$	$[\![def]\!]$	$[\![def]\!]$	$[\![def]\!]$
$[\![def]\!]$	$[\![e[iv \leftarrow [1,1]]\!]$	$[\![e[iv \leftarrow [1,2]]\!]$	$[\![e[iv \leftarrow [1,3]]\!]$
$[\![def]\!]$	$[\![e[iv \leftarrow [2,1]]\!]$	$[\![e[iv \leftarrow [2,2]]\!]$	$[\![e[iv \leftarrow [2,3]]\!]$
$[\![def]\!]$	$[\![e[iv \leftarrow [3,1]]\!]$	$[\![e[iv \leftarrow [3,2]]\!]$	$[\![e[iv \leftarrow [3,3]]\!]$
$[\![def]\!]$	$[\![def]\!]$	$[\![def]\!]$	$[\![def]\!]$

Fig. 11. The `genarray`-variant of the WITH-loop array comprehension

are defined by the second expression, the so-called *default expression.* The *shape expression* (i.e. the first expression after the key word `genarray`) must evaluate to a non-negative integer vector. The example WITH-loop in Fig. 11, hence, defines a matrix with 5 rows and 4 columns.

The middle part of the WITH-loop, the *partition* (line 2 in Fig. 11), defines a rectangular index subset of the defined array. A partition consists of a *generator* and an *associated expression.* The generator defines a set of index vectors along with an *index variable* representing elements of this set. Two expressions, which must evaluate to non-negative integer vectors of the same length as the value of the shape expression, define lower and upper bounds of a rectangular range of index vectors. For each element of this index vector set defined by the generator, the associated expression is evaluated with the index variable instantiated according to the index position. In the case of the `genarray`-variant the resulting value defines the element value at the corresponding index location of the array.

The default expression itself is optional with an element type dependent default default value, i.e. the fitting variant of zero (`false`, `'\0'`, `0`, `0f`, `0d` for types `bool`, `char`, `int`, `float`, `double`, respectively). If possible the compiler adds the appropriate value. A default expression may not even be needed if the generator already covers the entire index set.

The second WITH-loop-variant is the `modarray`-variant illustrated in Fig. 12. While the partition (line 2) is syntactically and semantically equivalent to the `genarray`-variant, the definition of the array's shape and the default rule for element values that are not contained in the generator-defined index set are different. The key word `modarray` is followed by a single expression. The newly defined array takes its shape from the value of that expression, i.e. we define an array that has the same shape as a previously defined array. Likewise, the element values at index positions not covered by the generator are obtained from the corresponding elements of that array. It is important to note that the `modarray`-WITH-loop does not destructively overwrite the element values of the existing array, as it would be common in the imperative world. Since SAC is a purely functional language, we semantically define a new array value that lives aside the existing one.

```
B = with {
      ([1,1] <= iv < [4,4]) : e(iv);
    }: modarray( A);
```

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]
[3,0]	[3,1]	[3,2]	[3,3]
[4,0]	[4,1]	[4,2]	[4,3]

$\Longrightarrow$

$[\![A[[0,0]]]\!]$	$[\![A[[0,1]]]\!]$	$[\![A[[0,2]]]\!]$	$[\![A[[0,3]]]\!]$
$[\![A[[1,0]]]\!]$	$[\![e[iv \leftarrow [1,1]]]\!]$	$[\![e[iv \leftarrow [1,2]]]\!]$	$[\![e[iv \leftarrow [1,3]]]\!]$
$[\![A[[2,0]]]\!]$	$[\![e[iv \leftarrow [2,1]]]\!]$	$[\![e[iv \leftarrow [2,2]]]\!]$	$[\![e[iv \leftarrow [2,3]]]\!]$
$[\![A[[3,0]]]\!]$	$[\![e[iv \leftarrow [3,1]]]\!]$	$[\![e[iv \leftarrow [3,2]]]\!]$	$[\![e[iv \leftarrow [3,3]]]\!]$
$[\![A[[4,0]]]\!]$	$[\![A[[4,1]]]\!]$	$[\![A[[4,2]]]\!]$	$[\![A[[4,3]]]\!]$

Fig. 12. The `modarray`-variant of the WITH-loop array comprehension

The third WITH-loop-variant supports the definition of reduction operations. It is characterised by the key word `fold` followed by the name of an eligible reduction function or operator and the neutral element of that function or operator. For certain built-in functions and operators the compiler is aware of the neutral element, and an explicit specification can be left out. SAC requires fold functions or operators to expect two arguments of the same type and to yield one value of that type. Fold functions must be associative and commutative. These requirements are stronger than in other languages with explicit reductions (e.g. `foldl` and `foldr` in many mainstream functional languages). This is motivated by the absence of an order on the generator defined index subset and ultimately by the wish to facilitate parallel implementations of reductions.

```
B = with {
      ([1,1] <= iv < [4,4]) : e(iv);
    }: fold( ⊕, neutr);
```

[1,1]	[1,2]	[1,3]
[2,1]	[2,2]	[2,3]
[3,1]	[3,2]	[3,3]

$$\Longrightarrow \quad \begin{aligned} [\![neutr]\!] &\oplus [\![e[iv \leftarrow [1,1]]]\!] \oplus [\![e[iv \leftarrow [1,2]]]\!] \oplus [\![e[iv \leftarrow [1,3]]]\!] \\ &\oplus [\![e[iv \leftarrow [2,1]]]\!] \oplus [\![e[iv \leftarrow [2,2]]]\!] \oplus [\![e[iv \leftarrow [2,3]]]\!] \\ &\oplus [\![e[iv \leftarrow [3,1]]]\!] \oplus [\![e[iv \leftarrow [3,2]]]\!] \oplus [\![e[iv \leftarrow [3,3]]]\!] \end{aligned}$$

Fig. 13. The `fold`-variant of the WITH-loop array comprehension

Note that the SAC compiler cannot verify associativity and commutativity of user-defined functions. It is the programmer's responsibility to ensure these properties. Using a function or operator in a `fold`-WITH-loop acts as an implicit assertion of the required properties. To be precise, neither floating point nor integer machine arithmetic is strictly speaking associative. It is up to the programmer to judge whether or not overflow/underflow in integer computations or numerical stability issues in floating point computations are relevant. If so and the exact order in which a reduction is performed does matter, the `fold`-WITH-loop is not the right choice. Instead, sequential loops as in C should be used. This is not a specific problem of SaC, but is owed to parallel reduction in general. The same issues appear in all programming environments that support parallel reductions, e.g. the reduction clause in OpenMP[24,25] or the collective operations in Mpi[26].

$$
\begin{array}{lcl}
\textit{WithLoopExpr} & \Rightarrow & \textbf{with} \ \left\{ \ \textit{Partition} \ \right\} \ : \ \textit{Operator} \\[1ex]
\textit{Partition} & \Rightarrow & \textit{Generator} \ : \ \textit{Expr} \ ; \\[1ex]
\textit{Generator} & \Rightarrow & \texttt{(} \ \textit{Expr RelOp Identifier RelOp Expr} \ \texttt{)} \\[1ex]
\textit{RelOp} & \Rightarrow & \texttt{<=} \ | \ \texttt{<} \\[1ex]
\textit{Operation} & \Rightarrow & \textbf{genarray} \ \ \texttt{(} \ \textit{Expr} \left[\ , \ \ \textit{Expr} \right] \ \texttt{)} \\
 & | & \textbf{modarray} \ \ \texttt{(} \ \textit{Expr} \ \texttt{)} \\
 & | & \textbf{fold} \ \ \texttt{(} \ \textit{FoldOp} \left[\ , \ \ \textit{Expr} \right] \ \texttt{)} \\[1ex]
\textit{FoldOp} & \Rightarrow & \textit{Identifier} \ | \ \textit{BinOp}
\end{array}
$$

Fig. 14. Formal definition of the (simplified) syntax of WITH-loop expressions

3.4 With-Loop Examples

Following the rather formal introduction of WITH-loops in the previous section we now illustrate the concept and its use by a series of examples. For instance, the matrix

$$
A = \begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 \\
30 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\
40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49
\end{pmatrix}
$$

can be defined by the following WITH-loop:

```
A = with {
      ([0,0] <= iv < [5,10]) : iv[[0]] * 10 + iv[[1]];
    }: genarray( [5,10]);
```

Note here that the generator variable `iv` denotes a 2-element integer vector. Hence, the scalar index values need to be extracted through selection prior to computing the new array's element value.

The following `modarray`-WITH-loop defines the new array `B` that like `A` is a 5×10 matrix where all inner elements equal the corresponding values of `A` incremented by 50 while the remaining boundary elements are obtained from `A` without modification:

```
B = with {
     ([1,1] <= iv < [4,9]) : A[iv] + 50;
    }: modarray( A);
```

This example WITH-loop defines the following matrix:

$$B = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 19 \\ 20 & 71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 & 29 \\ 30 & 81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 & 39 \\ 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \end{pmatrix}$$

Last not least, the following `fold`-WITH-loop computes the sum of all elements of array `B`:

```
sum =  with {
         ([0,0] <= iv < [5,10]) : B[iv];
       }: fold( +, 0);
```
which yields 2425.

3.5 Advanced Aspects of With-Loops

So far, we have focussed on the principles of WITH-loops and restricted ourselves to a simplified view. In fact, WITH-loops are much more versatile; Fig. 15 defines the complete syntax that we now explain step by step.

We begin with a major extension: a WITH-loop may have multiple partitions instead of a single one. With multiple partitions, disjoint index subsets of an array may be computed according to different specifications. For example, the WITH-loop

```
A = with {
      ([0,0] <= iv < [5, 8]) : iv[[0]] * 10 + iv[[1]];
      ([0,8] <= iv < [5,10]) : iv[[0]] + iv[[1]];
    }: genarray( [5,10], 0);
```
yields the matrix

$$A = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 9 & 10 \\ 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 10 & 11 \\ 30 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 11 & 12 \\ 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 12 & 13 \end{pmatrix}$$

where the left 8 columns are defined according to the first partition and the right 2 columns according to the second partition. One question that immediately arises when defining multiple partitions is what happens if the index sets defined by the generators are not pairwise disjoint. Since this question is generally undecidable for the compiler, we define that the in textual order last partition that covers a certain index defines the corresponding value.

WithLoopExpr	⇒	**with** { [*Partition*]+ } : *OperatorList*
Partition	⇒	*Generator* [*Block*] : *ExprList* ;
Generator	⇒	(*Bound RelOp GenVar RelOp Bound* [*Filter*])
Bound	⇒	*Expr* | .
RelOp	⇒	<= | <
GenVar	⇒	*Identifier*
	|	*IdentifierVector*
	|	*Identifier* = *IdentifierVector*
IdentifierVector	⇒	[[*Identifier* [, *Identifier*]*]]
Filter	⇒	**step** *Expr* [**width** *Expr*]
ExprList	⇒	*Expr* [, *Expr*]*
OperatorList	⇒	*Operator*
	|	(*Operator* [, *Operator*]*)
Operator	⇒	**genarray** (*Expr* [, *Expr*])
	|	**modarray** (*Expr*)
	|	**fold** (*FoldOp* [, *Expr*])
FoldOp	⇒	*Identifier* | *BinOp*

Fig. 15. Formal definition of the full syntax of WITH-loop-expressions

As in the previous example, it is often handy to access the scalar elements of the generator variable directly, instead of explicitly selecting elements inside the associated expression:

```
A = with {
      ([0,0] <= [i,j] < [5, 8]) : i * 10 + j;
      ([0,8] <= [i,j] < [5,10]) : i + j;
    }: genarray( [5,10]);
```

In fact, one can even use the generator variable in vector and scalar form in the same partition.

A significant extension of all WITH-loop variants concerns the generators. Rather than defining dense rectangular index spaces, extended generators may also define sparse periodic patterns of indices. For example, the WITH-loop

```
A = with {
      ([0,0] <= [i,j] < [5,10] step [1,2]) : i * 10 + j;
    }: genarray( [5,10], 0);
```

yields the matrix

$$\begin{pmatrix} 0\,0 & 2\,0 & 4\,0 & 6\,0 & 8\,0 \\ 10\,0 & 12\,0 & 14\,0 & 16\,0 & 18\,0 \\ 20\,0 & 22\,0 & 24\,0 & 26\,0 & 28\,0 \\ 30\,0 & 32\,0 & 34\,0 & 36\,0 & 38\,0 \\ 40\,0 & 42\,0 & 44\,0 & 46\,0 & 48\,0 \end{pmatrix}$$

An additional *width* specification allows generators to define generalised periodic grids as in the following example where

```
A = with {
      ([0,0] <= iv < [5,10] step [4,4] width [2,2]) : 9;
      ([0,2] <= iv < [5,10] step [4,4] width [2,2]) : 0;
      ([2,0] <= iv < [5,10] step [4,1] width [2,1]) : 1;
    }: genarray( [5,10]);
```

yields

$$\begin{pmatrix} 9&9&0&0&9&9&0&0&9&9 \\ 9&9&0&0&9&9&0&0&9&9 \\ 1&1&1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1&1 \\ 9&9&0&0&9&9&0&0&9&9 \end{pmatrix}$$

Expressions that define step and width vectors must evaluate to positive integer vectors of the same length as the other vectors of the generator. The full range of generators can be used with all WITH-loop variants.

In order to give a formal definition of index sets, let a, b, s, and w denote expressions that evaluate to appropriate vectors of length n. Then, the generator

(a `<= iv <` b `step` s `width` w)

defines the following set of index vectors:

$$\{\ iv \mid \forall_{j \in \{0,\ldots,n-1\}} : a_j \leq iv_j < b_j \ \wedge \ \ (iv_j - a_j) \ \bmod \ s_j < w_j \ \} \quad .$$

The last major extension concerns the operator. Actually, WITH-loops may come with a list of operators, and a single WITH-loop may combine multiple variants. For instance the WITH-loop

```
mini, maxi = with {
               ([0,0] <= iv < [5,10]) : A[iv], A[iv];
             }: (fold( min), fold( max));
```

simultaneously defines the minimum and the maximum value of the previously defined array `A`. Each generator is associated with a comma-separated list of expressions that correspond to the comma-separated list of operators. As this example illustrates, it is often handy to have the generator-associated expressions be preceded by a local block of assignments to abstract away complex or common subexpressions. Hence, the above example could also be written as

```
mini, maxi = with {
               ([0,0] <= iv < [5,10]) {
                                        a = A[iv];
                                      }: a, a;
             }: (fold( min), fold( max));
```

In practice, WITH-loops are often much simpler than they could be. Quite commonly they define homogeneous array operations where all elements of the index space are treated in the same way: a single generator covers the whole index space. To facilitate specification of the common case, dots may replace the bound expressions in generators. A dot as lower bound represents the least and a dot as upper bound represents the greatest legal index vector with respect to the shape vector of a `genarray`-WITH-loop or the shape of the referenced array in a `modarray`-WITH-loop. The lack of a reference shape restricts this feature in the case of a `fold`-WITH-loop to the lower bound. Here, a dot represents a vector of zeros with the same length as the vector defining the upper bound.

4 Programming Methodology

So far we have introduced the most relevant language features of SaC. In this section, we explain the methodology of programming in SaC, i.e. how the language features can be combined to write actual programs. We begin with the array type system of SaC (Section 4.1) and proceed to explain overloading (Section 4.2) and user-defined types (Section 4.3). At last, we explain the two major software engineering principles advocated by SaC: the principle of abstraction (Section 4.4) and the principle of composition (Section 4.5).

4.1 Array Type System

In Section 2 we introduced the basic types mostly adopted from C (i.e. `int`, `float`, `double`, `char` and `bool`). In Section 3 we discussed how to create arrays, but we carefully avoided any questions regarding the exact type of some integer matrix or double vector. We catch up with this deficit now.

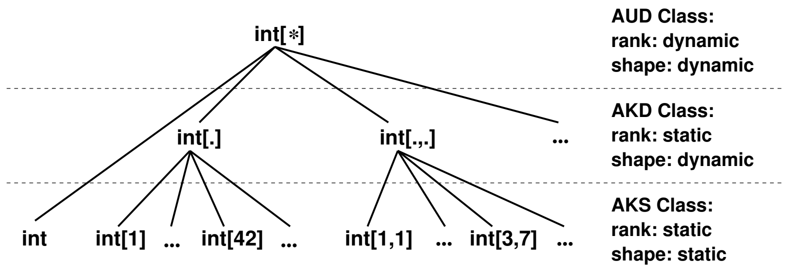


Fig. 16. The SaC array type system with the subtyping hierarchy

While SaC is monomorphic in scalar types including the base types of arrays, any scalar type immediately induces a hierarchy of array types with subtyping. Fig. 16 illustrates this type hierarchy for the example of the base type `int`. The shapely type hierarchy has three levels characterised by different amounts of compile time knowledge about shape and rank. On the lowest level of the subtyping hierarchy (i.e. the most specific types) we have complete compile time knowledge on the structure of an array: both rank and shape are fixed. We call this class *AKS* for *array of known shape.*

On an intermediate level of the subtyping hierarchy we still know the rank of an array, but abstract from its concrete shape. We call this class *AKD* for *array of known dimension.* For example, a vector of unknown length or a matrix of unknown size fall into this category. Note the special case for arrays of rank zero (aka scalars). Since there is only one vector of length zero, the empty vector, the shape of a rank-zero array is automatically known and the type `int[]` is merely an uncommon synonym for `int`.

Each type hierarchy also has a most common supertype that neither prescribes shape nor rank at compile time. We call such types *AUD* for *array of unknown dimension.* The syntax of array types is motivated by the common syntax for regular expressions: the Kleene star in the AUD type stands for any number of dots, including none.

4.2 Overloading

SaC supports overloading with respect to the array type hierarchy. The example in Fig. 17 shows three overloaded instances of the subtraction operator, one for 20×20-matrices, one for matrices of some shape and one for arrays of any rank and shape. As usual in subtyping there is a monotony restriction. For any two instances F_1 and F_2 of some function F with the same number of parameters and the same base types for each parameter either each parameter type of F_1 is a subtype of the corresponding parameter type of F_2 or vice versa. Function instances with different numbers of parameters are distinguished syntactically and there is no such restriction.

```
int[20,20] (-) (int[20,20] A, int[20,20] B) {...}
int[.,.]   (-) (int[.,.]   A, int[.,.]   B) {...}
int[*]     (-) (int[*]     A, int[*]     B) {...}
```

Fig. 17. Overloading with respect to the array type hierarchy

If necessary, function applications are dynamically dispatched to the most specific instance available. For example, if we apply the subtraction operator, under the definition of Fig. 17, to two integer matrices of unknown shape (AKD class), we can statically rule out the third instance because the second instance fits and is more specific. However, we can not rule out the first instance as the argument matrices at runtime could turn out to be of shape 20×20 and then the more specific first instance must be preferred over the more general second instance.

4.3 User-Defined Types

SaC supports user-defined types in very much the same way as many other languages: any type can be abstracted by a name. Following our general design principles, SaC adopts the C syntax for type definitions. For example, a type `complex` for complex numbers can be defined as a two-element vector by the following type definition:

```
typedef double[2] complex;
```

This type definition induces a further complete subtyping hierarchy with overloading. In contrast to C, however, SaC user-defined types are real data types and not just type synonyms. Values require explicit conversion between the defining type and the defined type or vice versa. Such conversions use the familiar syntax of C type casts. In fact, this notation is mainly intended as an implementation vehicle for proper conversion functions. Fig. 18 illustrates programming with user-defined types by an excerpt from the standard library's module for complex arithmetic.

```
typedef double[2] complex;

complex toc( double real, double imag)
{
  return (complex) [real, imag];
}

double real( complex cpx)
{
  return ((double[2])cpx)[[0]];
}

double imag( complex cpx)
{
  return ((double[2])cpx)[[1]];
}

complex (+) (complex a, complex b)
{
  return toc( real(a) + real(b), imag(a) + imag(b));
}

complex (*) (complex a, complex b)
{
  return toc( real(a) * real(b) - imag(a) * imag(b),
              real(a) * imag(b) + imag(a) * real(b));
}
```

Fig. 18. Basic definitions for complex numbers: type definition, conversion functions making use of the type cast notation and overloaded definitions of arithmetic operators based on the conversion functions introduced before

A few restrictions apply to user-defined types. The defining type must be an AKS type, i.e. another scalar type or a type with static shape, as in the case of type `complex` defined above. We have been working on removing this restriction and supporting truly nested arrays, i.e. arrays where the elements are again arrays of different shape and potentially different rank. For now, however, this is an experimental feature of SaC; details can be found in [27].

4.4 The Principle of Abstraction

As pointed out in Section 3.2 SaC only features a very small set of built-in array operations. Commonly used aggregate array operations are defined in SaC itself in a completely generic way. Although not built-in, aggregate array operations are applicable to arguments of any rank and shape. A prerequisite for this design are the shape-generic programming capabilities of WITH-loops. As introduced in Sections 3.3 and 3.5, all relevant syntactic positions of WITH-loops may host arbitrary expressions. In the examples so far we merely used constant vectors for the purpose of illustration. In practice, WITH-loops are key to shape- and rank-generic definitions of array operations.

Fig. 19 demonstrates the transition from a shape-specific implementation over a shape-generic implementation to a rank-generic implementation taking element-wise subtraction of two arrays as a running example. The first (overloaded) instance of the subtraction operator is defined for 20×20 integer matrices. It makes use of a single WITH-loop and essentially maps the built-in scalar subtraction operator to all corresponding elements of the two argument arrays. As the shape of the matrix is fixed, we can simply use constant vectors in the syntactic positions for result shape, lower bound and upper bound.

Of course, it is neither productive nor elegant or even possible to explicitly overload the subtraction operator for each potential argument array shape. The second instance in Fig. 19 sticks to the two-dimensional case, but abstracts from the concrete size of argument matrices. This generalisation immediately raises an important question: how to deal with argument arrays of different shape? There are various plausible answers to this question, and the solution adopted in our example is to compute the element-wise minimum of the shape vectors of the two argument arrays. With this solution we safely avoid out-of-bound indexing while at the same time restricting the function domain as little as possible. The resulting vector `shp` is used both in the shape expression of the `genarray`-WITH-loop and as upper bound in the generator. Since indexing in SaC always starts at zero, we can stick to a constant vector as lower bound. Note that the generator-associated expression remains unchanged from the shape-specific instance.

One could argue that in practice, it is very rare to encounter problems that require more than 4 dimensions, and, thus, we could simply define all relevant operations for one, two, three and four dimensions. However, for a binary operator that alone would already require the definition of 16 instances. Hence, it is of practical relevance and not just theoretical beauty to also abstract from the rank of argument arrays, not only the shapes, and to support fully rank-generic programming.

```
int[20,20] (-) (int[20,20] A, int[20,20] B)
{
  res = with {
          ([0,0] <= iv < [20,20]) : A[iv] - B[iv];
        }: genarray( [20,20], 0);
  return res;
}

int[.,.] (-) (int[.,.] A, int[.,.] B)
{
  shp = min( shape(A), shape(B));
  res = with {
          ([0,0] <= iv < shp) : A[iv] - B[iv];
        }: genarray( shp, 0);
  return res;
}

int[*] (-) (int[*] A, int[*] B)
{
  shp = min( shape(A), shape(B));
  res = with {
          (0*shp <= iv < shp) : A[iv] - B[iv];
        }: genarray( shp, 0);
  return res;
}
```

Fig. 19. Three overloaded instances of the subtraction operator for arrays of known shape (AKS, top), arrays of known dimension (AKD, middle) and arrays of unknown dimension (AUD, bottom)

The third instance of the subtraction operator in Fig. 19 demonstrates this further abstraction step. Apart from using the most general array type `int[*]`, the rank-generic instance is surprisingly similar to the rank-specific one. The main issue is an appropriate definition of the generator's lower bound, i.e. a vector of zeros whose length equals that of the shape expression. We achieve this by multiplying the shape vector with zero.

So far, we expected argument arrays to be at least of the same shape. With a rank-generic type, however, we must also consider argument arrays of different rank. What would happen if we apply the subtraction operator to a 10-element vector and a 5×5-matrix? The shapes of the argument arrays are, consequently, `[10]` and `[5,5]`, respectively. Assuming an implementation of the minimum function along the lines of the subtraction operator discussed here, we obtain `[5]` as the minimum of the two vectors. Thus, the WITH-loop defines a 5-element vector whose elements are homogeneously defined as the subtraction of the corresponding elements from the argument arrays `A` and `B`. Since `A` is a vector and we select using a 1-element index vector, selection yields a scalar. As array `B` is a matrix, selection with a 1-element index vector yields a (row) vector. As a consequence, the subtraction in the body of the WITH-loop does not refer to the built-in scalar subtraction, but recursively back to the rank-generic instance. Whereas the type of this instance suggests to support a scalar and a vector argument, the definition inevitably leads to non-terminating recursion. We can easily avoid this by defining two more overloaded instances of the subtraction operator that cover the cases where one argument is scalar, as shown in Fig. 20.

```
int[*] (-) (int A, int[*] B)
{
  shp = shape(B);
  res = with {
          (0*shp <= iv < shp) : A - B[iv];
        }: genarray( shp, 0);
  return res;
}

int[*] (-) (int[*] A, int B)
{
  shp = shape(A);
  res = with {
          (0*shp <= iv < shp) : A[iv] - B;
        }: genarray( shp, 0);
  return res;
}
```

Fig. 20. Additional overloaded instances of the subtraction operator as they are found in the SaC standard library

It is one of the strengths of SaC that the exact behaviour of array operations is not hard-wired into the language definition. This sets SaC apart from all other languages with dedicated array support. Alternative to our above solution with the minimum shape, one could argue that any attempt to subtract two argument arrays of different shape is a programming error as in Fortran-90 or Apl. The same could be achieved in SaC by comparing the two argument shapes and raising an exception if they differ. The important message here is that SaC does not impose a particular solution on its users: anyone can provide an alternative array module implementation with the desired behaviour.

A potential wish for future versions of SaC is support for a richer type system, in which shape relations like equality can be properly expressed in the array types. For example, matrix multiplication could be defined with a type signature along the lines of

```
double[a,c] matmul( double[a,b] X, double[b,c] Y)
```

This leads to a system of dependent array types that we have studied in the context of the dependently typed array language Qube [28,29]. However, how to carry these ideas over to SaC in the presence of overloading and dynamic dispatch requires a plethora of future research.

4.5 The Principle of Composition

The generic programming examples of the previous section open up an avenue to define a large body of array operations by means of WITH-loops. For instance, Fig. 21 shows the definition of a generic convergence check. Two argument arrays `new` and `old` are deemed to be convergent if for every element (reduction with logical conjunction) the absolute difference between the new and the old value is less than a given threshold `eps`.

```
bool is_convergent (double[*] new, double[*] old, double eps)
{
  shp = min( shape(new), shape(old));
  res = with {
          (. <= iv < shp) : abs(new[iv] - old[iv]) < eps;
        }: fold( &&);
  return res;
}
```

Fig. 21. Rank-generic definition of a convergence check

While defining the convergence check as in Fig. 21 is a viable approach, it lacks a certain elegance: we indeed re-invent the wheel with the minimum shape computation, that is actually only needed for the element-wise subtraction, for which we have already solved the issue with the code shown in Fig. 20. A closer look into the WITH-loop quickly reveals that we deal with a computational pipeline of basic operations on array elements. This can be much more elegantly and concisely expressed following the other guiding software engineering principle in SAC: the *principle of composition.*

```
bool is_convergent (double[*] new, double[*] old, double eps)
{
  return all( abs( new - old) < eps);
}
```

Fig. 22. Programming by composition: specification of a generic convergence check

As demonstrated in Fig. 22, the compositional specification of the convergence check is entirely based on applications of predefined array operations from the SAC standard library: element-wise subtraction, absolute value, element-wise comparison and reduction with Boolean conjunction. This example demonstrates how application code can be designed in an entirely index-, loop-, and comprehension-free style.

Ideally the use of WITH-loops as versatile but accordingly complex language construct would be confined to defining basic array operations like the ones used in the definition of the convergence check. And, ideally all application code would solely be composed out of these basic building blocks. This leads to a highly productive software engineering process, substantial code reuse, good readability of code and, last not least, high confidence into the correctness of programs. The case study on generic convolution developed in Section 5 further demonstrates how the principle of composition can be applied in practice.

5 Case Study: Convolution

In this section we illustrate the ins and outs of SaC programming by means of a case study: convolution. Following a short introduction to the algorithmic principle (Section 5.1) we show a variety of implementations of individual convolution steps that illustrate the principles of abstraction and composition. (Sections 5.2–5.5). Finally, we extend our work to an iterative process (Section 5.6).

5.1 Algorithmic Principle

Convolution follows a fairly simple algorithmic principle. Essentially, we deal with a regular, potentially multidimensional grid of data cells, as illustrated in Fig. 23. Convolution is an iterative process on this data grid: in each iteration (often referred to as *temporal dimension* in contrast to the *spatial dimensions* of the grid) the value at each grid point is recomputed as a function of the existing old value and the values of a certain neighbourhood of grid points. This neighbourhood is often referred to as *stencil*, and it very much characterises the convolution.

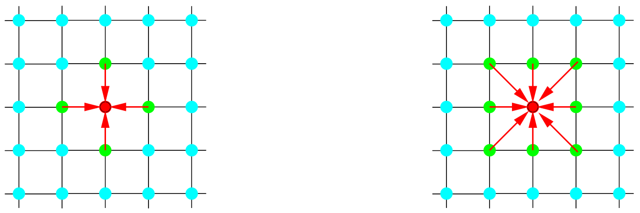


Fig. 23. Algorithmic principle of convolution, shown is the 2-dimensional case with a 5-point stencil (left) and a 9-point stencil (right)

In Fig. 23 we show two common stencils. With a *five-point stencil* (left) only the four direct neighbours in the two-dimensional grid are relevant. By including the four diagonal neighbours we end up with a *nine-point stencil* (right) and so on. In the context of cellular automata these neighbourhoods are often referred to as *von Neumann neighbourhood* and *Moore neighbourhood*, respectively. With higher-dimensional grids, we obtain different neighbourhood sizes, but the principle can straightforwardly be carried over to any number of dimensions.

Since any concrete grid is finite, boundary elements leaves essentially two choices: *cyclic boundary conditions* and *static boundary conditions*. In the former case the neighbourship relation is defined round-robin, i.e., the left neighbour of the leftmost element is the rightmost element and vice versa. In the latter case the grid is surrounded by a layer of elements that remain constant throughout the convolution process.

In principle, any function from a set of neighbouring data points to a single new one is possible, but in practice variants of weighted sums prevail. The algorithmic principle of convolution has countless applications in image processing, computational sciences, etc.

5.2 Convolution Step with Cyclic Boundary Conditions

As a first step in our case study of implementing various versions of convolution in SaC, we restrict ourselves to nearest neighbours and to the arithmetic mean of these neighbour values as the compute function, i.e. a weighted sum where all weights are identical. Furthermore, we use cyclic boundary conditions for now and leave static boundary conditions for later. With these fairly simple convolution parameters, we aim at shape- and rank-generic SaC implementations that are based on the software engineering principles of abstraction and composition. Whenever possible we employ an index-free programming style that treats arrays in a holistic way rather than as loose collections of elements.

```
double[.] convolution_step (double[.] A)
{
  R = A + rotate( 1, A) + rotate( -1, A);
  return R / 3.0;
}
```

Fig. 24. 1-dimensional index-free convolution step

With the code example in Fig. 24 we start with an index-free and shape- but not rank-generic implementation of a single convolution step. The function `convolution_step` expects a vector of double precision floating point numbers and yields a (once) convolved such vector. The implementation is based on the `rotate` function from the SaC standard library. It rotates a given vector by a certain number of elements towards ascending or descending indices. Rotation towards ascending indices means moving the rightmost element of the vector (the one with the greatest index) to the leftmost index position (the one with the least index). This implements cyclic boundary conditions almost for free. Adding up the original vector, the left-rotated vector and the right-rotated vector yields the convolved vector. The only task left is the division of each element by 3.0 to obtain the arithmetic mean. This implementation of 1-dimensional convolution makes use of a total of five data-parallel operations: two rotations, two element-wise additions and one element-wise division.

```
double[*] convolution_step (double[*] A)
{
  R = A;
  for (i=0; i<dim(A); i++) {
    R = R + rotate( i, 1, A) + rotate( i, -1, A);
  }
  return R / tod( 2 * dim(A) + 1);
}
```

Fig. 25. Rank-generic convolution step

We now generalise the one-dimensional convolution to the rank-generic convolution shown in Fig. 25. We use the same approach with rotation towards ascending and descending indices, but now we are confronted with a variable number of axes along which to rotate the argument array. We solve the problem by using a `for`-loop over the number of dimensions of the argument array `A`, which we obtain through the built-in function `dim`. In each dimension we rotate `A` by one element towards ascending and towards descending indices. Here, we use an overloaded, rank-generic version of the `rotate` function that takes the rotation axis as the first argument in addition to the rotation offset and the array to be rotated as second and as third argument, respectively.

The original argument array and the various rotated arrays are again summed up as in the one-dimensional solution. To eventually compute the arithmetic mean we still need to divide array `R` by the number of arrays we summed up. This number can easily be obtained through the `dim` function, as shown in Fig. 25. Since the SaC standard library currently restricts itself to defining arithmetic operators on identical argument types, we must explicitly convert the resulting integer to double using the conversion function `tod`. Of course, we could extend the standard library by all kinds of type combinations, but we refrain from this for two reasons. Firstly, it would substantially increase the size of the corresponding module due to combinatorial explosion. Secondly, it would reduce the programmer's reflection on the types involved.

5.3 Convolution Step with Static Boundary Conditions

With a rank-generic, index-free convolution step for cyclic boundary conditions at hand we aim at carrying over these ideas to the case of static boundary conditions. For didactic purposes we again begin with the one-dimensional case shown in Fig 26. While the signature of the `convolution_step` function remains as before, we now consider only the inner elements of the argument array `A` to be proper grid points and all boundary elements to form the constant halo.

Implementation-wise, we simply replace the applications of the rotate function in the code of Fig. 24 by corresponding applications of the `shift` function. The `shift` function is very similar to the `rotate` function with the exception that vector elements are not moved round-robin. Instead, elements moved out of the vector on one side are discarded while default values are moved into the vector

```
double[.] convolution_step (double[.] A)
{
  conv  = (A + shift( 1, A) + shift( -1, A)) / 3.0;

  inner = tile( shape( conv) - 2, [1], conv);
  res   = embed( inner, [1], A);

  return res;
}
```

Fig. 26. 1-dimensional convolution step with static boundary conditions

from the other side. The default value in the version of `shift` used here is zero; other overloaded variants of the `shift` function in the SaC standard library allow the programmer to explicitly provide a default value.

Unlike the cyclic boundary case, however, we are not yet done with the computation in line 3. Treating all arrays in a holistic way, that computation includes the boundary elements of the arrays in the computation. This is algorithmically wrong as the halo elements shall remain constant throughout all iterations. To achieve this, we explicitly "correct" the boundary elements in lines 5 and 6. We do this by first creating the array of all inner elements (i.e. the "real" grid points) and then embedding this array within the original array `A`. We make use of two more functions from the SaC standard array library:

– `double[*] tile( int[.]` *`shp`*`, int[.]` *`offset`*`, double[*]` *`array`*`)` yields the subset of *`array`* of shape *`shp`* beginning at index *`offset`*;
– `double[*] embed( double[*]` *`small`*`, int[.]` *`offset`*`, double[*]` *`big`*`)` yields an array of the same shape as *`big`*. The elements are those of *`big`* except for the elements from index *`offset`* onwards for the shape of *`small`* which are taken from *`small`*.

In Fig. 27 we generalise the one-dimensional convolution kernel with static boundary conditions to a rank-generic implementation. We adopt the same approach as in the case of cyclic boundary conditions in Section 5.2 and make use of a `for`-loop over the rank of the argument array. The 3-ary, multidimensional variant of the `shift` function is an extension of the 2-ary, one-dimensional function used so far that is fully analogous to the corresponding extension of the `rotate` function used previously.

The correction of the boundary elements can be carried over from the one-dimensional to the multidimensional case with almost no change, thanks to the rank-invariant definitions of the library functions `tile` and `embed`. The only modification stems from the need to use a vector of ones whose length equals the rank of the argument array. For any rank-specific implementation we could simply use the corresponding vector constant as in Fig. 26, but for a rank-generic solution we need a small trick: we multiply the shape vector of the argument array by zero, which yields an appropriately sized vector of zeros, and then add one to obtain the desired vector of ones.

```
double[*] convolution_step (double[*] A)
{
  conv = A;
  for (i=0; i<dim(A); i++) {
    conv = conv + shift( i, 1, A) + shift( i, -1, A);
  }
  conv = conv / tod( 2 * dim(A) + 1));

  vector_of_ones = shape(conv) * 0 + 1;
  inner = tile( shape(conv) - 2, vector_of_ones , conv);
  res   = embed( inner, vector_of_ones , A);

  return res;
}
```

Fig. 27. Rank-generic convolution step with static boundary conditions

5.4 Red-Black Convolution

An algorithmic variant of convolution is called *red-black* convolution. In red-black convolution the grid is bipartite with each grid point either belonging to the red or to the black set. Convolution is then computed alternatingly on the red and on the black grid points while the other values are simply carried over from the previous iteration. Typically, the red and black sets are not randomly distributed over the index set of the grid, but themselves follow some regular alternating pattern along one or multiple axes.

```
double[*] redblack_step (bool[*] mask, double[*] A)
{
  A = where(  mask, convolution_step( A), A);
  A = where( !mask, convolution_step( A), A);

  return A;
}
```

Fig. 28. Red-black convolution

Fig. 28 shows a highly generic SaC implementation of a red-black convolution step where the choice of red and black grid points is abstracted into an additional parameter in form of a Boolean mask. We consecutively apply the `convolution_step` function to the red and to the black elements by restricting its effect using the `where` function from the SaC standard library:

– `double[*] where( bool[*]` *`mask`*`, double[*]` *`then`*`, double[*]` *`else`*`)` yields the array of the same shape as the Boolean array *`mask`* whose elements are taken from the corresponding elements of array *`then`* where the mask is true and from *`else`* where not.

The where function resembles the FORTRAN-90 language construct of the same name: Our implementation of red-black convolution can easily be combined with static and with cyclic boundary conditions.

5.5 Stencil-Generic Convolution

In all examples so far we have anticipated a direct-neighbour stencil, i.e., we had two neighbours in the one-dimensional case, four neighbours in the two-dimensional case, six neighbours in the three-dimensional case and so on. In this final escalation step we aim at abstracting from the concrete shape of the stencil and support arbitrary dynamic neighbourhoods. We return to cyclic boundary conditions for simplicity, but the idea for correcting the boundary elements for static boundary conditions, as introduced in Section 5.3 can be carried over straightforwardly.

```
double[*] convolution_step (double[*] A, double[*] weights)
{
  R = with {
        ( 0*shape(weights) <= iv < shape(weights) ) :
          weights[iv] * rotate( shape(weights)/2-iv, A);
      } : fold( +);
  return R;
}
```

Fig. 29. Neighbourhood-generic, rank-generic convolution step

The convolution_step function shown in Fig. 29 is parameterised over a multidimensional array of weights. Although the type system of SAC does not allow us to express this restriction formally, we anticipate that the argument array A and the array weights have the same rank. For example, let us consider to convolve a matrix. Then the weight matrix

$$\begin{pmatrix} 0.0 & 0.2 & 0.0 \\ 0.2 & 0.2 & 0.2 \\ 0.0 & 0.2 & 0.0 \end{pmatrix}$$

would represent the 5-point stencil that we have used so far. The weight matrix allows us to easily define any neighbourhood and, of course, to give different weights to different neighbourship relations. The weight array is also not restricted to three elements per axis; we could easily include neighbourship relations including the left-left neighbour, etc.

The algorithmic idea behind the code in Fig. 29 is a generalisation of the approach taken so far using a for-loop over the rank of the argument array. We use a WITH-loop over the shape of the weight array. For instance, the weight matrix above would induce a 3×3 index space for the fold-WITH-loop. For each element of this index space (i.e. for each element of the weight array) we rotate the argument array into the direction of that weight's position in the array of weights. Returning to the above example, we rotate the argument array one element up and one element left for the upper left element of the weight matrix. For the central element of the weight matrix (index `[1,1]`) we do not rotate the argument at all, etc.

```
double[*] rotate( int[.] offsets, double[*] A)
{
  for (i=0; i < min( shape(offsets)[0], dim(A))) {
    A = rotate( i, offsets[i], A);
  }

  return A;
}
```

Fig. 30. Generically defined multidimensional rotation

The alert reader will have noticed that the rotation function used in Fig. 29 is again an overloaded variant of the rotation functions used so far; we show its definition in Fig. 30. This function makes use of a vector of rotation offsets. The first element of the offset vector determines the rotation offset along the first axis of the argument array and so on. Accordingly, we use a `for`-loop over the minimum of the length of the offset vector and the argument array rank. For each rotation offset we apply the previous version of `rotate` on the corresponding array axis. If the offset vector length is less than the argument array rank, trailing axes of the argument array remain unrotated; surplus offsets are ignored. We end up with a total of nine rotated and weighted arrays. The `fold`-WITH-loop eventually sums them up using the overloaded element-wise plus operator on arrays, which yields the convolved array.

5.6 Multiple Convolution Steps

Until now we have only looked at individual convolution steps. Convolution, however, is an iterative process of such steps. In the simplest case the number of iterations is given and thus known a-priori. Fig. 31 shows the SaC implementation of this scenario: we simply employ a `for`-loop to repeatedly apply individual convolution steps to the grid.

Computing an a-priori known number of convolution steps is a typical benchmark situation. In practice, it is often relevant to continue with the convolution until a certain fixed point is reached, i.e. continue until for no grid point the current iteration's value differs from the previous iteration's value by more than a given threshold. Fig. 32 shows our SaC implementation. As the number of convolution steps to be performed is a-priori unknown, we use a `do-while`-loop.

```
double[*] convolution (double[*] A, int iter)
{
  for (i=0; i<iter; i++) {
    A = convolution_step( A);
  }

  return A;
}
```

Fig. 31. Convolution with given number of iteration steps

Of course, the argument array could in principle meet the convergence criterion right away, which would call for a `while`-loop instead of a `do-while`-loop, but we consider this a pathological case and, hence, stick to the `do-while`-loop. As the loop predicate we use the `is_convergent` function introduced in Section 4.5. The convergence test needs to refer to both the old and the new version of the data grid, hence we introduce the local variable `A_old`.

```
double[*] convolution (double[*] A, double eps)
{
  do {
    A_old = A;
    A = convolution_step( A_old);
  }
  while (!is_convergent( A, A_old, eps));

  return A;
}
```

Fig. 32. Convolution with convergence test

6 Case Study: Differentiation

Our second case study looks into numerical differentiation along one or two axes. We begin with simple SaC definitions (Section 6.1). They motivate a language extension of SaC called the *axis control notation* (Section 6.2). Finally, we apply this notation to numerical differentiation (Section 6.3).

6.1 Differentiation in 1 and 2 Dimensions

In its simplest form numerical differentiation is based on a function, given as a vector of function values, and the constant difference between two argument values. The first derivation of the function is then defined as a vector that is one element shorter than that representing the function itself. The values are the differences between neighbouring function values divided by their distance.

```
double[.] dfDx( double[.] vec, double delta)
{
  return ( drop( [1], vec) - drop( [-1], vec) ) / delta;
}
```

Fig. 33. 1-dimensional numerical differentiation

Fig. 33 shows a straightforward implementation of 1-dimensional differentiation as a SAC function `dfDx`. Rather than making use of a WITH-loop for its definition we follow the SAC methodology and apply the principles of abstraction (Section 4.4) and composition (Section 4.5). The principle of abstraction mainly materialises itself in the form of the function `drop` that we introduce alongside its counterpart, the `take` function:

– `double[*] drop( int[.]` *`dv`*`, double[*]` *`a`*`)`
  drops as many elements from each axis of the argument array *a* as given by the *drop vector* *dv*. The first element of the drop vector determines how many elements to drop from the outermost axis of the array and so on. Positive drop values drop leading elements while negative drop values drop trailing elements. If the length of the drop vector exceeds the rank of the array, excess drop values are ignores; if the drop vector is shorter than the rank of the array, trailing axes of the array remain untouched. Dropping more elements than an array has along any axis results in zero elements alongside that axis and in an overall empty result array.
– `double[*] take( int[.]` *`tv`*`, double[*]` *`a`*`)`
  takes as many elements from each axis of the argument array *a* as given by the *take vector* *tv*. All small prints are equivalent to those of the `drop` function.

```
double[.,.] dfDy( double[.,.] mat, double delta)
{
  return with {
           (. <= xv <= .): dfDx( mat[xv], delta);
         }: genarray( take( [1], shape( mat)));
}

double[.,.] dfDx( double[.,.] mat, double delta)
{
  return transpose( dfDy( transpose( mat), delta));
}
```

Fig. 34. 2-dimensional numerical differentiation

In the same way as a unary function can be represented as a vector of function values, a binary function can be represented as a matrix of values. This gives us two directions for numerical differentiation, typically referred to as x and y. Fig. 34 shows SAC functions to differentiate a binary function with respect to the first parameter (`dfDx`) and the second parameter (`dfDy`).

Differentiating a binary function with respect to the second parameter means computing differences alongside the inner dimension of the matrix `mat`. In other words we interpret the matrix as a column vector of row vectors, apply our 1-dimensional differentiation function `dfDx` (Fig. 33) to each row vector and end up with a column vector of derivatives. Differentiating a binary function with respect to the first parameter (`dfDx` in Fig. 34) could be achieved in a similar way, but we instead advertise the SAC methodology and define it based on the `dfDy` function and matrix transposition.

6.2 Axis Control Notation

The definition of the function `dfDy` in the previous section represents a common pattern in array programming that can be generalised as a three step process:

1. interpret a rank k array as an array of rank m of (equally shaped) arrays of rank n with $m + n = k$;
2. individually apply some function to each of the inner arrays;
3. laminate the partial results to form the overall result array.

Fig. 35 illustrates this pattern by a 3-dimensional example. We start with a $4 \times 4 \times 4$-cube of elements. We then (re-)interpret this cube as 4×4-matrix of 4-element vectors, apply some function individually to each of the 16 vectors and laminate the 16 result vectors back into a $4 \times 4 \times 4$-cube.

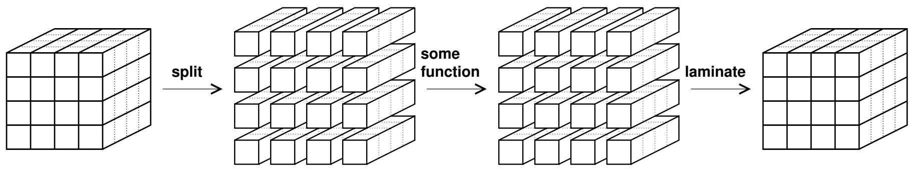


Fig. 35. The split-compute-laminate algorithmic pattern

In Fig. 35 we assume the function to be *uniform*, i.e. shape-preserving. This is not required, and Fig. 36 illustrates the split-compute-laminate principle with a reduction operation. In this example we interpret the $4 \times 4 \times 4$-cube as a vector of four 4×4-matrices. In the second step each matrix is individually reduced to a scalar value, and in the third step these scalar values are laminated into a 4-element vector. While compute functions do not need to preserve the shape of

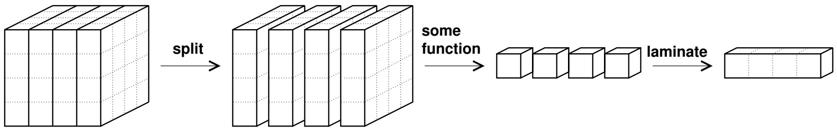


Fig. 36. The split-compute-laminate pattern with non-uniform function

the argument, they are nonetheless restricted: the shape of the result must not depend on anything but the shape of the argument.

SaC provides specific support for the common algorithmic pattern of split-compute-laminate through the *axis control notation* [30]. We sketch out the two syntactic extensions in Fig. 37. First, we extend array selection such that an index vector may contain dots instead of expressions. Semantically, a dot in an index vector means to select all elements in the corresponding dimension of the array selected from. This extension allows us to select entire subarrays of an array not only in trailing dimensions (as with index vectors that are shorter than the array's rank), but in any choice of dimension. Note that vectors containing dots are not first-class values, but are exclusively permitted in index position.

```
Expr          ⇒ ...
              | Expr [ SelVec ]
              | { FrameVec -> Expr }

SelVec        ⇒ [ DotOrExpr [ , DotOrExpr ]* ]

FrameVec      ⇒ Id | [ DotOrId [ , DotOrId ]* ]

DotOrExpr     ⇒ . | Expr

DotOrId       ⇒ . | Id
```

Fig. 37. Syntax of axis control notation

The other extension shown in Fig. 37 is an expression in curly brackets that defines a particular mapping from a set of indices represented by the vector left of the arrow to a set of values defined by the expression on the right hand side of the arrow. The extent of the index set is implicitly derived from the corresponding variables appearing in index position in the right hand side expression.

We illustrate the axis control notation the very concise definition of the `transpose` function shown in Fig. 38. The frame vector `[i,j]` defines a 2-dimensional index space whose boundaries are given by the reversed shape of `mat` through `i` and `j` appearing in index position on the right hand side. A complete account of the axis control notation can be found in [30].

```
double[.,.] transpose( double[.,.] mat)
{
  return {[i,j] -> mat[[j,i]]};
}
```

Fig. 38. Definition of matrix transpose with axis control notation

6.3 Differentiation with Axis Control Notation

In this section we demonstrate how the definitions of our differentiation functions from Section 6.1 benefit from the axis control notation as shown in Fig. 39. Both functions, `dfDY` and `dfDx`, clearly benefit from the axis control notation that enables much conciser and more readable definitions. In particular, the definitions now expose the same symmetries as the underlying mathematical problem they implement. In `dfDy` we interpret the argument matrix as a column vector of row vectors and apply 1-dimensional differentiation to each row vector. In `dfDx` we interpret the argument matrix as row vector of column vectors and accordingly apply 1-dimensional differentiation to each column vector.

```
double[.,.] dfDy( double[.,.] mat, double delta)
{
  return {[i,.] -> dfDx( mat[[i,.]], delta)};
}

double[.,.] dfDx( double[.,.] mat, double delta)
{
  return {[.,j] -> dfDx( mat[[.,j]], delta)};
}
```

Fig. 39. 2-dimensional differentiation with axis control notation

7 Modules

This section introduces the module system of SaC, that provides the necessary features for programming-in-the-large. Since SaC only provides very few built-in operations, the SaC standard library with its extensive support for high-level array operations is instrumental for writing even short programs. Hence, some familiarity with the module system is essential. We start with introducing the concept of name spaces (Section 7.1). We proceed with explaining several ways of making symbols from other name spaces available and discuss their differences in the context of function overloading (Sections 7.2 and 7.3). At last, we show how to write modules and make symbols available to other name spaces (Section 7.4).

7.1 Name Spaces

Name spaces are a common mechanism to resolve name clashes when symbols with the same name are defined in different modules of an application. In SaC every module defines a name space. Any program (featuring a `main` function), adds a name space `main`. In a multi-module application any symbol can uniquely be identified by its qualified name consisting of the name space and the symbol name connected by a double colon (as in C++).

```
double, double sincos( double val)
{
  return (Math::sin(val), Math::cos(val));
}
```

Fig. 40. Using symbols from other name spaces with qualified identifiers

Fig. 40 shows a simple example: we define a function `sincos` that simultaneously yields the sine and the cosine of a given value by applying the corresponding individual functions from the SaC standard library, more precisely the `Math` module. Functions `sin` and `cos` are identified by their qualified names.

7.2 The Use Directive

Qualified names quickly become unhandy if symbols are frequently used, in particular when names are not as short as in the previous example. Therefore, SaC supports ways to automatically resolve name spaces and let the compiler generate qualified identifiers internally. The programmer, still, needs to define a search space for the compiler to look for symbols. By means of the directive

`use` *`name_space`* `all;`

preceding all definitions in a module/program all symbols defined in the given module are made known locally. With this technique our `sincos` function can be re-written as in Fig. 41.

```
use Math: all;

double, double sincos( double val)
{
  return (sin(val), cos(val));
}
```

Fig. 41. Making all symbols from another module available in the current name space with the `use` directive

Symbols must not have multiple definitions within the search space as that would make their resolution ambiguous. An exception are functions with different argument counts or different argument base types. In the presence of overloading such functions are considered different symbols.

Nonetheless, more stringent control over which symbols to make available from what modules is required in practice. Therefore, the key word `all` in the `use`-directive can be replaced by a comma-separated list of identifiers embraced in curly brackets. Alternatively, the key words `all except` followed by a list of symbols allows us to explicitly exclude a set of named symbols from the search space. Fig. 42 illustrates these features by a further variation of the running example. We now explicitly choose the symbols `sin` and `tan` from the `Math` module while all other math support comes from an alternative `FastMath` module.

```
use Math: {sin, tan};
use FastMath: all except {sin, tan};

double, double sincos( double val)
{
  return (sin(val), cos(val));
}
```

Fig. 42. Making specific symbols from different name spaces available in the current name space with the qualified `use` directive

Functions can only be added to or removed from the symbol search space by their name. The module system does currently not distinguish overloaded instances of a function based on the number or the types of parameters.

7.3 The Import Directive

The `use`-directive adds symbols to the search space of the SAC compiler. While very handy in practice, it needs to be used with some care in order to avoid ambiguities (and thus compiler error messages) in the resolution of function symbols. Such an ambiguity arises whenever the same function name is defined in two name spaces, and both are *used* from a third name space. If the two function definitions differ in the number or the base types of parameters, function applications in the third name space can still be disambiguated. In contrast, purely shapely overloading can generally not be resolved at compile time, but warrants a runtime decision. Combining multiple shapely overloaded instances of the same function across name space boundaries, thus, may change the meaning of a function a-posteriori, potentially violating the intentions of the developers of the original modules. Therefore, we disallow *using* shapely overloaded functions.

Nonetheless, shapely overloading across module boundaries when used correctly and consciously, can be a very powerful mechanism, and the `import`-directive supports exactly this. Whereas the `use`-directive makes symbols from another name spaces accessible in the current name space, the `import`-directive clones symbols from other name spaces in the current name space. As a consequence, the compiler constructs a completely new dispatch tree that takes all *imported* instances as well as the locally defined instances equally into account.

```
import foomod: {foo};

int[42] foo( int[42] x) { ... }
int[.] foo( int[.] x) { ... }

int bar( int[*] a)
{
  ...
  b = foo( a);
  ...
}
```

Fig. 43. Shapely overloading across name spaces with the `import`-directive

Fig. 43 illustrates this be means of a small example. We first *import* the potentially already overloaded definition of `foo` from the name space `foomod`. Afterwards, we further overload the function `foo` with two more definitions, one for integer vectors of length 42 and one for integer vectors of arbitrary length. When we dispatch the application of `foo` in the body of function `bar` all instances of `foo` are equally considered, regardless whether they are locally defined or imported. The `import`-directive supports the same syntactic variations as the `use`-directive; both directives can be freely interspersed.

7.4 Defining Modules

A SaC module differs from a program in two aspects: the absence of a `main` function and a module header consisting of the key word `module`, the module name and a semicolon. Fig. 44 shows a simple example. We pick our convolution case study from Section 5 up and provide a module `Convolution` defining two overloaded instances of a function `convolution` computing either a fixed number of iterations or a variable number of iterations with convergence check as in Figs. 31 and 32, respectively. Before actually defining the new function instances, however, we need to make a number of functions defined elsewhere available that we need to define `convolution`, e.g. the various implementations of individual convolution steps or the convergence check. And of course, we require the basic array support from the standard library. The choice of selective or general use or import of symbols into the current name space is mainly motivated to showcase the various syntactic options.

The most interesting aspect of a module is the question which symbols are made available outside and which are kept hidden within the module . Two directive, `provide` and `export`, give programmers fine-grained control over this question. By default any symbol defined in a module is only accessible in the module itself. The `provide` directive makes symbols available to be *used* in other name spaces; the `export` directive makes symbols available for both *use* or *import*. Thus, the owner of a module decides whether or not functions can be shapely overloaded later with all consequences on semantics. The `provide`

```
module Convolution;

use Array: all;
import ConvolutionStep: all;
use Convergence: {is_convergent};

provide all except {convolution};
export {convolution};

double[*] convolution (double[*] A, int iter)   {...}
double[*] convolution (double[*] A, double eps) {...}
```

Fig. 44. Example of a module implementation that bundles the two generic convolution functions developed in Section 5.6

and `export` directives support the same features for symbol selection as the corresponding `use` and `import` directives.

In the example of Fig. 44 we export the two instances of the `convolution` function. More for didactic purposes we choose also to provide all other symbols defined in the current module. Note that we import (not use) the symbols from module `ConvolutionStep`. As a consequence, they are cloned in the current module and hence can be provided as genuine symbols of module `Convolution`. Again, our example rather illustrates the various options our module system provides; in practice one would rather only provide the convolution functions.

8 Input and Output

In this chapter we sketch out the principles of SaC's support for input/output in particular and for stateful computations in general. We begin with the user perspective on basic file I/O (Section 8.1), then shown how imperative-appearing I/O constructs can safely be integrated into the functional context of SaC (Section 8.2) and conclude with a complete I/O example with proper error checking and handling (Section 8.3).

8.1 Basic File I/O

Integration of I/O facilities into SaC is guided by two seemingly conflicting design principles. On the one hand, we aim at extending to look-and-feel of C programming to I/O-related SaC code; on the other hand, it is crucial to retain SaC's status as a pure functional language on the semantic level and not to restrain any optimisation potential.

Just as with the language kernel, programmers with a background in imperative programming should not be bothered by the conceptual troubles of manipulating the state of devices in a state-free environment. We certainly do not want our programmers to familiarise themselves with theoretically demanding concepts such as monads [31,32] and uniqueness types [33,34]. And we definitely do not want to rely on our programmers being experts in category theory to write a *hello world* program in SaC. Instead, any C programmer should be able to write I/O-related code in SaC even without the need to learn a new API.

```
import StdIO: all;
import ArrayIO: all;

int main()
{
  a = 42;
  b = [1,2,3,4,5];

  errcode, outfile = fopen( "filename", "w");

  fprintf( outfile, "a = %d\n", a);
  fprint( outfile, b);

  fclose( outfile);

  return 0;
}
```

Fig. 45. Example for doing file input/output in SaC

Fig. 45 shows a simple file I/O example written in SaC. For now, we ignore all potential semantic issues of the code and merely emphasise the similarities between SaC and C proper. First, we import all symbols from the two relevant SaC modules of the standard library. In the `main` function we begin with opening a file using a clone of C's `fopen` function. Like its C counterpart `fopen` expects two character strings as arguments. The first denotes the name of the file to be opened, and the second determines the file mode. In the example, we open the file `filename` for writing. The supported file modes are identical with C proper. In fact, the SaC `fopen` function is merely a wrapper for the C `fopen` function called through SaC's foreign language interface [35].

Unlike C, the SaC `fopen` function makes use of the support for multiple return values and yields two values: a file handle (`outfile`) and an error code (`errcode`). For the sake of simplicity, we expect the opening of the file to succeed and ignore the error code for now. We will discuss a complete example with proper error checking in Section 8.3.

Having opened the file, we write some text and a scalar value to the file using the `fprintf` function, that again is a clone of the corresponding C function. Next we write an entire array to the file using the SaC-specific function `fprint`. Since the C `fprintf` family of functions have no support for array-related conversion specifiers, we add the `fprint` family of functions for array output. Finally, we close the file using the usual `fclose` function. The example demonstrates how well we achieve our first aim: supporting I/O in way that is familiar to C programmers.

8.2 Imperative I/O vs Functional Semantics

Right now, it seems much less clear how we achieve our second aim: functionally sound I/O. After all, many I/O functions do not even yield a value and would be dead code in a purely functional interpretation. Worse, the textual order of statements now matters: there is an implicit execution order not enforced by data dependencies. The solution is surprisingly simple, nonetheless. In analogy to the interpretation of C-style loops as syntactic sugar for tail recursion code like in Fig. 45 is nothing but an imperative illusion of a purely functional code. Fig. 46 shows its *functional interpretation*. In essence, the compiler automatically establishes the necessary data dependencies that describe the intended execution order in a functionally sound way.

```
FileSystem, int main( FileSystem theFileSystem)
{
  a = 42;
  b = [1,2,3,4,5];

  theFileSystem, errcode, outfile
    = fopen( theFileSystem, "file_name", "w");

  outfile = fprintf( outfile, "a = %d\n", a);
  outfile = fprint( outfile, b);

  theFileSystem = fclose( theFileSystem, outfile);

  return (theFileSystem, 0);
}
```

Fig. 46. Functional interpretation of the I/O code in Fig. 45, compiler-inserted intermediate code typeset in italics

The `main` function has an extended signature: it now receives a representation of the file system and yields, in addition to the usual integer return code, a potentially modified representation of the file system. Likewise, all I/O-related functions in the body of `main` receive additional arguments and yield additional values. The `fopen` function takes the file system as an additional argument and yields a modified file system (representation). Assuming opening of the file succeeds, the new file system differs from the old file system in exactly this property: the named file was closed and is now open for writing.

The two output functions `fprintf` and `fprint` take the file handle as before, but additionally return the file handle. This creates a data dependency from the call to `fopen` over `fprintf` and `fprint` to the final closing of the file by `fclose`. In analogy to `fopen`, the `fclose` function takes the file system as an additional parameter and yields a modified file system, in which the file is no longer open but closed. This final state of the file system is eventually returned to the execution environment.

The SaC compiler actually does these transformations to deal with a proper functional representation of code when it comes to optimisation. Conceptually and technically, our solution is based on a variant of *uniqueness types* [34,36] as developed for I/O in the functional language Clean. The types of `theFileSystem` and `outfile` are uniqueness types, and one can easily verify that every definition (left hand side use) of one these variables has exactly (at most) one reference (right hand side use). The main difference to uniqueness types in Clean lies in the fact that the entire conceptual complexity of dealing with state in a functional context is hidden in a number of modules from the SaC standard library (and of course corresponding compiler support). Actually doing I/O in an application program is as simple as in imperative languages while under the hood everything is safe and clean. The non-expert programmer does not need to understand the ins and outs of safe functional I/O.

The SaC compiler does check the uniqueness property, but for the normal user it is close to impossible to produce a uniqueness violation. As long as a programmer merely makes use of the various I/O modules of the SaC standard library, the automatic (internal) expansion of code along the lines of Fig. 46 prior to uniqueness checking almost inevitably leads to correct code. Thus, programmers are usually not bothered with cryptic uniqueness-related error messages. In some cases the uniqueness checker can, however, detect common programming errors, e.g. missing or repeated closing of files. A more complete coverage of SaC I/O can be found in [36].

A particular issue is the combination of input/output, where a particular execution order is important, with SaC's data-parallel WITH-loops, where a concrete execution order is deliberately not guaranteed, nor even defined. Due to space limitations we refer the interested reader to [37] for a comprehensive discussion of this aspect of SaC.

8.3 File I/O with Error Checking

In our initial I/O example in Fig. 45 we deliberately skipped all error checking and crossed fingers that opening the file succeeds. We now extend the simple file I/O example with proper error checking making use of the `SysErr` module from the standard library. This module essentially replaces C's `errno` variable. Fig. 47 shows the complete example.

We remember that the `fopen` function yields an error code in addition to the file handle Unlike in Fig. 45, we now check this error code before making use of the file handle. The `fail` function discriminates success codes from failure

```
import StdIO: all;
import ArrayIO: all;
import SysErr: all;

int main()
{
  a = 42;
  b = [1,2,3,4,5];

  errcode, outfile = fopen( "file_name", "w");

  if (fail(errcode)) {
    fprintf( stderr, "%s\n", strerror( errcode));
  }
  else {
    fprintf( outfile, "a = %d\n", a);
    fprint( outfile, b);

    fclose( outfile);
  }

  return 0;
}
```

Fig. 47. Complete I/O example with error checking

codes and yields a Boolean value suitable for use in a predicate. Upon failure we print a message to `stderr`. Just as in C, `stdout`, `stdin` and `stderr` are file handles that are always open for writing or reading. In SaC they are so-called *global objects*: stateful entities that follow the same visibility and scoping rules as functions. They can be accessed anywhere in function bodies and are subject to use/import from other name spaces and provide/export to other name spaces. For more information on global objects we refer the interested reader to [36]. The `strerror` function is identical to its C counterpart and yields a problem description in the string form. Note that we do not close the file in the first branch as we have not (successfully) opened it either. The second branch is analogous to Fig. 45.

9 Foreign Language Interfaces

This section describes SaC's foreign language interfaces. They allow SaC code to interoperate with existing or yet to be developed C code. Two such interfaces exist that are equally important in practice: the c4sac interface allows SaC code to call C functions (Section 9.1) while the sac4c interface supports the compilation of SaC modules such that they can be embedded within larger C applications (Section 9.2).

9.1 Calling C from SAC

The c4sac interface is an indispensable feature for making SAC communicate with the outside world. Most I/O functions introduced in Section 8 are merely SAC wrappers for the corresponding C functions. It would not only be very cumbersome to re-implement I/O support from scratch in SAC, we would also inevitably re-invent the wheel and simply waste engineering effort. Instead, we aim at reusing existing implementations as much as possible and seek to excel in core areas of SAC.

```
external double cos( double x);
  #pragma linkwith "m"
  #pragma linksign [0,1]

external double, double sin_cos( double x);
  #pragma linksign [1,2,3]
  #pragma linkobj "src/Math/mymath"

external syserr, File fopen( string filename, string mode);
  #pragma linkobj "src/File/fopen.o"
  #pragma effect theFileSystem
  #pragma linkname "SACfopen"
  #pragma linksign [0,1,2,3]

external int, ... scanf( string format);
  #pragma linkobj "src/File/scanf.o"
  #pragma effect stdin

external void fprint( File &file,
                      int dim, int[.] shp, int[*] array)
  #pragma refcounting [4]
```

Fig. 48. Examples of foreign function declarations from the SAC standard library

Fig. 48 illustrates the c4sac foreign language interface by three examples from different modules of the SAC standard library. Foreign function declarations like the ones in Fig. 48 may appear interspersed with SAC function definitions throughout SAC modules. In principle, a pure declaration starting with the key word `external` followed by standard SAC function header and terminated by a semicolon suffices. In practice, the SAC compiler often needs some more information to seamlessly integrate an imperative foreign function into the functional world of SAC. Several pragmas serve this purpose.

Our first example is the `cos` function from SAC's `Math` module, obviously a foreign declaration for the corresponding `cos` function from the C math library `libm`. Most math functions are easy targets for SAC's foreign language interface as they directly expose a functional interface computing a new scalar value from an existing scalar value with call-by-value parameter passing. Nonetheless, someone needs to tell the SAC compiler to link an executable program with `libm` as soon as the `cos` function is used anywhere. This is done with the `linkwith`-pragma.

The `linksign`-pragma is more complex. It describes a mapping of SAC parameters and results to those of the corresponding C function. A vector defines for each result and parameter in textual order from left to right onto which position of the C function it is mapped. This pragma allows us to map C functions that return multiple values through reference parameters into proper SAC functions with multiple return values. The numbers in the vector stand for the positional parameters of the corresponding C function, where zero represents the explicit return position.

So, in the case of the `sin` function the `linksign`-pragma merely specifies the expected, i.e. the SAC function is mapped to a C function with the same function type. To illustrate the `linksign`-pragma Fig. 48 also contains a foreign declaration of an artificial function `sin_cos` that simultaneously yields the sine and the cosine of a given value. While in SAC this can elegantly by expressed with two return types, there is no equivalent C type. The `linksign`-pragma makes the SAC compiler expect the existence of a C function

```
void sin_cos( double *sin, double *cos, double x)
```

and generate corresponding function calls. It is even possible to map one return value and one parameter onto the same parameter location of the C function. In this way C functions that take a pointer and manipulate the data behind the pointer can properly be used from SAC.

Unlike `sin` and `cos`, no function `sin_cos` is defined in `libm`. Hence, the SAC compiler needs to be informed where to locate code order to generate appropriate linker calls. This is the purpose of the `linkobj`-pragma.

The third declaration in Fig. 48 makes the `fopen` function, extensively discussed in Section 8.1, available in SAC. The `effect`-pragma tells the SAC compiler that this function makes an implicit side effect on the global object `theFileSystem`. This information is essential for the compiler to generate explicit data dependencies as shown in Section 8.2.

More adaptation between C and SAC is required due to the different approaches to error reporting. The SAC `fopen` function explicitly yields an error condition while the C `fopen` function yields `NULL` and sets the global `errno` variable. This difference requires a thin wrapper layer implemented in C. This wrapper can obviously not be named `fopen`. Thus, the `linkname`-pragma allows us to manipulate the name of the function that is actually called by the SAC compiler in place of `fopen`. The `linksign`-pragma again merely describes the default: the error condition is returned via the C function's result while the file handle is implemented as a reference parameter in the first parameter position and the other parameters follow in order.

The foreign declaration of the `scanf` function demonstrates how variable argument lists from C are mapped to SAC. Three dots on the left hand side indicate that `scanf` yields an unknown number of results in addition to the usual integer value returned by C's `scanf` function. These are always mapped to trailing reference parameters of the corresponding C function and, thus, to the expected type signature of `scanf` in C.

Our last example shows the declaration of a rank-generic print function for arrays. We use a function like this to implement the `fprint` function for arrays that we used throughout our examples in Section 8. Here, we expose the structural properties of the argument array explicitly. The ampersand marks the first parameter as a *reference parameter*. This triggers the addition of a corresponding result value as explained in Section 8.2. The `refcounting`-pragma declares the function to take care of reference counting for the fourth argument, i.e. the array to be printed. In this case the anticipated prototype of the C function changes such that 2 C parameters implement the one SAC parameter, the first being a pointer to the array itself (in flat, contiguous representation), the other being a pointer to an integer number that exposes the reference counter of the array. While this feature obviously requires in-depth understanding of the SAC memory management subsystem, it allows the expert user to safely implement destructive array updates and take similar advantages of reference counting as the SAC compiler itself does. In Section 10.4 we explain SAC memory management in greater detail.

With the facilities of the c4sac language interface we have made most of the standard C library functions from `libc` and `libm` available in SAC. These are extended by a range of array-specific functions implemented in C, e.g. for inout and output of arrays.

9.2 Calling SAC from C

Equally important to the c4sac interface, though for different reasons, is the sac4c interface that makes entire SAC modules available within otherwise C-implemented applications. Of course, we promote to use SAC to implement whole applications, but we must acknowledge that transition to SAC is substantially eased if programmers can choose to only implement parts of an application in SAC. This also allows us to concentrate on application aspects like compute-intensive kernels, for which SAC is tailor-made, and avoid engineering effort to be directed into directions that are not the core of our research, say for example support for GUI-based applications.

In principle, any standard SAC module can be used from C code, but it needs some mending to expose and publish a C-compatible interface. For this purpose we provide a separate tool as part of the SAC installation: `sac4c`. This tool takes a compiled SAC module as an argument and generates among others a C header file with the type and function declarations exposed by the module. Furthermore, `sac4c` generates the necessary linker information regarding all directly and indirectly needed SAC modules as well as all further object files dependent through the c4sac interface.

Fig. 49 shows a simple example of C code making use of the `Convolution` module defined in Section 7.4. At first, we include two header files, one providing the necessary generic declarations of the sac4c interface, the other being generated by the `sac4c` tool. In the example we compute 99 convolution steps

```
#include "sac4c.h"
#include "Convolution.h"

int main( int argc, char *argv[])
{
  double *matrix;
  SACarg *arg, *iter, *res;
  int rank;

  matrix = C_code_that_creates_matrix( 1024, 1024);

  SACinit( argc, argv);

  arg  = SACARGconvertFromDoublePointer( matrix, 2, 1024, 1024);
  iter = SACARGconvertFromIntScalar( 99);

  Convolution__convolution2( &res, arg, iter);

  rank = SACARGgetDim( res);
  assert( rank==2);

  matrix = SACARGconvertToDoubleArray( res);
  plot( matrix);

  SACfinalize();
  return 0;
}
```

Fig. 49. Example C code making use of the sac4c foreign language interface

on a 1024 × 1024 double precision floating point matrix. Before starting any computations, however, we must initialise the SaC runtime system by a call to the `SACinit` function; upon completion the runtime system should be shutdown by a call to `SACfinalize`.

At the center of Fig. 49 we can identify the call to our SaC-implemented convolution function. The C function name has automatically been derived from the SaC module name and the SaC function name with two underscores in between (double underscores are not permitted in SaC identifiers). The trailing number 2 helps to resolve SaC function overloading with different arity and declares the function to be binary. The C prototype of the function is

```
void Convolution__convolution2(
  SACarg **res, SACarg *mat, SACarg *iter);
```

Functions made available through the sac4c interface are always `void`-functions and exchange arguments and results with the C world through a dedicated abstract type `SACarg`. This type helps us to expose SaC overloading both on base type and on shape to the C world. The sac4c interface comes with a range of functions that convert C arrays (contiguous, uniform chunks of memory) into SaC arrays and vice versa. Towards SAC the naked C pointer is equipped with SAC-style multidimensional shape information. On the way back structural properties of result arrays can be queried and, eventually, flat C arrays extracted for further analysis or processing. C arrays handed over to SAC must not be touched thereafter; C arrays obtained from SAC are guaranteed to be alias-free.

10 Compilation Technology

In this section we discuss the fundamental challenges of compiling SAC source code into competitive executable code for a variety of parallel computing architectures and outline how compiler and runtime system address these issues. Following an overview of the compiler architecture (Section 10.1) we concentrate on type inference and specialisation (Section 10.2), optimisation (Section 10.4) and code generation for various parallel architectures (Section 10.5). In place of a proper evaluation, Section 10.6 provides an annotated bibliography covering programmability, productivity and performance issues across a wide range of problems and target architectures.

10.1 The SAC Compiler at a Glance

Despite the intentional syntactic similarities, SAC is far from merely being a variant of C. SAC is a complete programming language, that only happens to resemble C in its look and feel. A fully-fledged compiler is needed to implement the functional semantics and to address a series of challenges when it comes to achieving high performance.

Fig. 50 shows the overall internal organisation of the SAC compiler `sac2c`. It is a many-pass compiler around a central slowly morphing abstract intermediate code representation. We chose this design to facilitate concurrent compiler engineering across multiple individuals and institutions. Today, we have around 200 compiler passes, and Fig. 50 only shows a macroscopic view of what is really going on behind the scenes. The SAC compiler, however, is very verbose with respect to its efforts: the interested programmer can stop compilation after any pass and have the internal representation printed as annotated SAC source code.

As a first step, usual lexicographic and syntactic analyses transform textual source code into an abstract syntax tree. All remaining compilation steps work on this internal representation, that is subject to a number of lowering steps. Over the years, we have developed a complete, language-independent compiler engineering tool suite, that has successfully been re-used in other projects [38] as well as in a series of courses on compiler construction at the University of Amsterdam. In the following, however, we leave out such engineering concerns and rather take a conceptual view on the SAC compilation process.

The first major code transformation shown in Fig. 50 is named *functionalisation.* Here, we turn the imperative(-looking) source code into a more functional representation. For instance, C-style branches turn into functional conditionals, and C-style loops become proper tail-recursive functions, as explained in

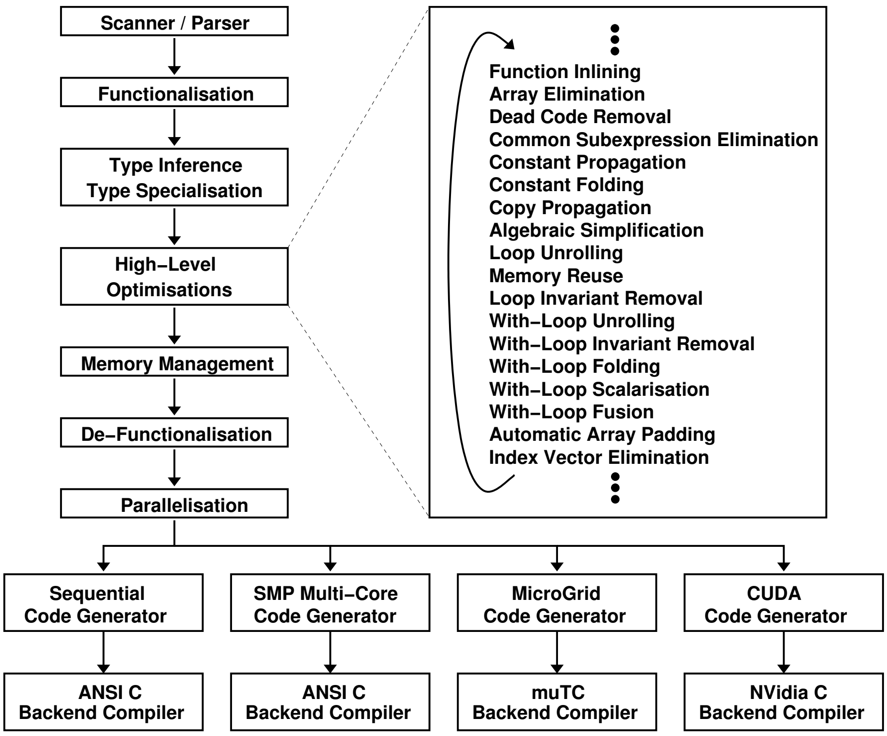


Fig. 50. Organisation of the compilation process

Section 2.2. Likewise, we augment state-based code with the missing data dependencies, as outlined in Section 8.2. All these transformations are eventually undone prior to code generation in a *de-functionalisation* step.

10.2 Type Inference and Specialisation

This part of the compiler implements the array type system outlined in Section 4.1. It annotates types to local variables and checks type declarations provided. Furthermore, the type inference system resolves function dispatch in the context of subtyping and overloading. Where possible function applications are dispatched statically; where necessary appropriate code is generated to make the decision at runtime. More information on this aspect of the SaC compiler can be found in [39].

The other important aspect handled by this part of the compiler is *function specialisation*. Shape- and rank-invariant specifications are a key feature of SaC. It is sort of obvious that the less we know at compile time about structural properties of arrays, the less efficiently will the generated code perform at runtime.

We are faced with the classical trade-off between abstraction and performance. Specific concerns are, for instance, how to generate code to operate on arrays whose rank is unknown at compile time and, hence, for whom no static nesting of loops can be derived, or how to generate efficient code from lists of generators when cache hierarchies demand arrays to be traversed in linear storage order.

From a software engineering point of view, all code should be written in a rank-generic (AUD) or at least shape-generic (AKS) way. From a compiler perspective, however, shape-specific code offers much better optimisation opportunities, can do with a much leaner runtime representation and generally shows considerably better performance. A common trick to reconcile abstract programming with high runtime performance is specialisation. In fact, the SAC compiler aggressively specialises rank-generic code to rank-specific (shape-generic) code and again shape-generic code into shape-specific code.

```
double[*] convolution (double[*] A, int iter)   {...}
double[*] convolution (double[*] A, double eps) {...}

specialize convolution (double[1024,1024] A, int iter)
specialize convolution (double[1024,1024] A, double eps)
```

Fig. 51. Helping the compiler with specialisation directives: while both instances of the convolution function (see Section 5.6) are defined in a rank-generic way, the compiler is advised to generate specialisations for 1024×1024-matrices

Specialisation can only be effective to the extent that rank and shape information is somehow accessible by the compiler. While `sac2c` makes every effort to infer the shapes needed, there are scenarios in which the required information is simply not available in the code. For instance, argument arrays could be read from files. Other common examples arise from any use of the sac4c foreign language interface described in Section 9.2. Hence, full compiler support for generating shape-generic and rank-generic executable code cannot be avoided.

More than occasionally, however, programmers do know or can at least make an educated guess as to which array shapes will be relevant at runtime. Specialisation directives, as shown in Fig. 51, allow us to give hints to the compiler which shapes will be relevant at runtime without compromising the rank- and shape-generic programming methodology and code base. The compiler creates the recommended specialisations in addition to those that it would generate by itself and transparently integrates them into the function dispatch mechanism.

The code in Fig. 51 completes our running example on convolution. Starting out in Section 5.6 we defined the two instances of the convolution function that either compute a given number of iterations or continue to iterate until a given convergence threshold is reached. In Section 7.4 we showed how these functions can be abstracted into a SAC module. We then demonstrated in Section 9.2 how this SAC module can be made available to be used from a C-implemented application. In the example of Fig. 49 we ran convolution on 1024×1024-matrices.

In this scenario the shape of the array to be convolved is statically known in the C code, but there is no way for the SaC compiler to know. Hence, it is forced to execute a rank-generic implementation of convolution, which is likely to deliver poor performance. The specialisation directives of Fig. 51 turn the tide and make the sac4c interface dynamically select the highly optimised binary convolution code for 1024×1024-matrices.

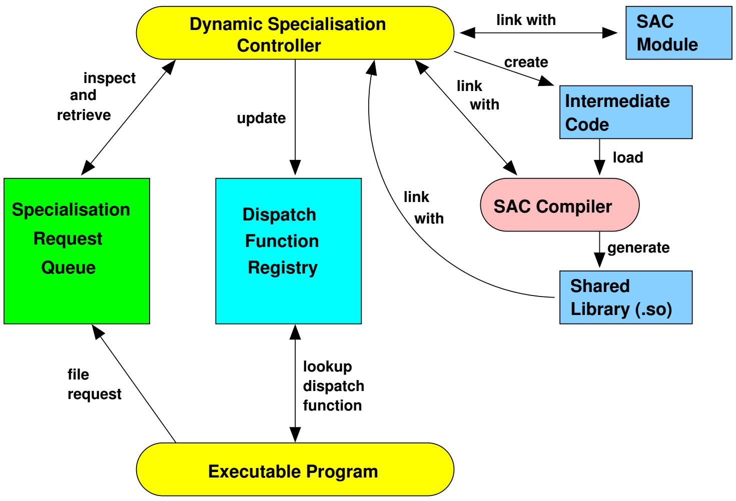


Fig. 52. Architecture of the SaC adaptive compilation framework

There are a number of scenarios, however, that rule out the helping hand of the programmer as well. For instance, the provider of some module and the user of that module could simply be distinct, or the nature of an application rules out the availability of shape information until the application is actually run. To address these cases we have devised the adaptive compilation infrastructure sketched out in Fig. 52.

The essential idea is to postpone specialisations until application runtime. When generic functions execute, they file a specialisation request that includes the full set of array shapes appearing in the concrete application. One or more *specialisation controllers* asynchronously take care of such requests, retrieve the partially compiled intermediate code from the corresponding SaC (binary) modules, run essential parts of the SaC compilation tool chain with all knowledge available at application runtime and eventually generate a specialised and highly optimised binary implementation of the function that initiated the request.

The running application is then dynamically linked with that new code, and the function dispatch mechanism is updated to include the new specialisation. When the same function is applied again to arguments of the same shapes, the specialised implementation will be chosen by the dispatch mechanism instead of the generic one. The whole approach is based on the (realistic) assumption that in many applications the number of different array shapes actually appearing is limited, even though at compile time no educated guess can be made on which shapes may and may not be relevant. A complete description and evaluation of our adaptive compilation framework can be found in [40]

10.3 High-level Optimisations

As apparent from a short glimpse at Fig. 50, high-level program optimisations constitute a major part of the SaC compiler and account for a substantial fraction of compiler engineering. Only the most prominent and/or relevant transformations are actually included in Fig. 50. They can coarsely be classified into two groups: (variations of) standard textbook optimisations and SaC-specific optimisations related to arrays.

The compositional programming methodology advocated by SaC creates a particular compilation challenge. Without dedicated compiler support it inflicts the creation of many temporary arrays at runtime, which adversely affects performance: large quantities of data must be moved through the memory hierarchy to perform almost negligible computations per array element. We quickly hit the memory wall and see our cores mainly waiting for data from memory rather than computing. With individual WITH-loops as basis for parallelisation, compositional specifications also incur high synchronisation and communication overhead.

As a consequence, the major theme of the array optimisation lies in condensing many light-weight array operations, more technically WITH-loops, into much fewer heavy-weight WITH-loops. Such techniques universally improve a number of ratios that are crucial for performance: the ratio between computations and memory operations, the ratio between computations and loop overhead and, in case of parallel execution, the ratio between computations and synchronisation and communication overhead. We identified three independent optimisation cases and address each one with a tailor-made program transformation:

– WITH-*loop-folding* [41] identifies computational pipelines where the result of one WITH-loop is referenced in a subsequent WITH-loop. If so, the reference in the second WITH-loop is replaced by the corresponding element definition from the first WITH-loop. Multi-generator WITH-loops and offset computations on index vectors make this a non-trivial undertaking. A good example is the convergence check in Fig. 22. Naive compilation would yield three temporary intermediate arrays before the final reduction is computed. WITH-loop-folding transforms the code into a single WITH-loop similar to the one shown in Fig. 21.
– WITH-*loop-fusion* [42] aims at WITH-loops that compute (data-)independent values based on a common or overlapping argument set. A typical example would be searching for the least and for the greatest value in an array. Naive compilation would yield one WITH-loop each, and the common argument array would be pumped through the entire memory hierarchy twice. WITH-loop-fusion, as the name suggests, fuses such WITH-loops into a

single multi-operator WITH-loop traversing argument arrays only once. For instance, WITH-loop-fusion manages to fuse the convolution step and the convergence check in Fig. 32 into a single WITH-loop after properly resolving the data dependency between the convolution step and the convergence check
- WITH-loop-scalarisation [43] joins nested WITH-loops, where the element value of an outer WITH-loop is itself a WITH-loop-defined array. Naive compilation would materialise a temporary array for each index of the outer WITH-loop. Arrays of complex numbers, for instance, lead to this situation as each complex number itself again is an array, i.e. a 2-element vector (see Section 4.3).

These optimisations are essential for making the compositional programming style advocated by SAC feasible in practice; a survey can be found in [44].

Other array-specific optimisations aim at avoiding the creation of small vectors used for indexing purposes (*index vector elimination* [45]) or optimise the cache utilisation in the context of densely stored multi-dimensional arrays (*array padding* [46]) to name just two. Moreover, the SAC compiler puts considerable effort into compiling complex generator sets of WITH-loops, potentially with multiple strided generators, etc, into an abstract representation that traverses the involved arrays in linear storage order whenever possible. This technique [47] is crucial to effectively utilise cache hierarchies essential for achieving good performance on modern systems.

The textbook optimisations first and foremost act as enablers of the array-specific optimisations. They create larger optimisation contexts (e.g. function inlining, loop unrolling), do all sorts of partial evaluation (e.g. constant folding and propagation, loop unrolling, algebraic simplification) or aim at avoiding superfluous computations (e.g. dead code removal, common subexpression elimination, loop invariant removal). While these optimisations are common in industrial-strength C compilers, the functional semantics of SAC allows us to apply them much more aggressively than what is possible in imperative environments.

10.4 Memory Management

Stateless arrays require memory resources to be managed automatically at runtime. This is a key ingredient of any functional language, and it is well understood how to design and implement efficient garbage collectors [48,49,50]. So, the stress here is rather on *arrays*. In serious applications arrays often require large contiguous chunks of memory, easily hundreds of MegaBytes and more. Such sizes require many design decisions in memory management to be reconsidered, e.g. they rule out copying garbage collectors.

On a more conceptual level we need to deal with the *aggregate update problem* [51]. Often an array is computed from an existing array by only changing a few elements. Or, imagine a recurrence relation where vector elements are computed in ascending index order based on their left neighbour. A straightforward functional implementation would need to copy large quantities of data unchanged from the “old” to the “new” array. As any imperative implementation would simply overwrite array elements as necessary, the functional code could never achieve competitive performance. Of course, one could also question the unboxed, dense in-memory representation silently assumed here, but this is likewise well known to be no solution.

As a domain-specific solution for array processing, SaC uses *non-deferred reference counting* [52] for garbage collection. Each array is augmented with a reference counter, and the generated code is likewise augmented with reference counting instructions that dynamically keep track of how many conceptual copies of an array exist. Compared with other garbage collection techniques non-deferred reference counting has the unique advantage that memory can immediately be reclaimed as soon as it turns into garbage. All other techniques in one way or another decouple the identification and reclamation of dead data from the last operation that makes use of the data.

Only non-deferred reference counting supports a number of optimisations that are crucial for achieving high performance in functional array programming. The ability to dynamically query the number of references of an array prior to some eligible operation creates opportunities for immediate memory reuse. Take for example a simple arithmetic operator overloaded for arrays like subtraction as discussed in Section 4.4. The definition of subtraction on arrays is point-wise and the result array requires exactly the same amount of memory as any of the two argument arrays. If one of them shows a reference counter value of one prior to computing subtraction, that argument array's memory can immediately be reused to store the result array. As a consequence, not only a costly memory allocation is avoided, but also the memory footprint of the operation is reduced by one third leading to much better cache hierarchy utilisation on typical cache-based computing systems.

In other cases we may not only be able to reuse memory but also to reuse the data already present in that memory. Consider a WITH-loop as in the following SaC code fragment:

```
b = with {
       (. <= iv < shape(a) / 2) : a[iv] + 1;
    }: modarray(a);
```

Here, an array `b` is computed from an existing array `a` such that the upper left corner (in the 2-dimensional case) is incremented by one while the remaining elements are copied from `a` proper. If we can reuse the memory of `a` to store `b`, we can effectively avoid to copy all those elements that remain the same in `b` as in `a`. Such techniques are important prerequisites to compete with imperative languages in terms of performance. A survey on SaC memory management can be found in [53].

Unlike other garbage collection techniques, non-deferred reference counting still relies on a heap manager for allocations and de-allocations. Standard heap managers are typically optimised for memory management workloads characterised by many fairly small chunks. In array processing, however, extremely large chunks are common, and they are often handled inefficiently by standard heap managers. Therefore, SaC comes with its own heap manager tightly integrated with compiler and runtime system and properly equipped for multithreaded execution [54].

10.5 Parallelisation and Code Generation

An important (non-coincidental) property of WITH-loops is that evaluation of the associated expression for any element of the union of index sets is completely independent of all others. This allows the compiler to freely choose any suitable evaluation order. We thoroughly exploit this property in the various WITH-loop-optimisations described above, but in the end the main motivation for this design is ease of parallelisation.

In contrast to auto-parallelisation in the imperative world, our problem is not to decide *where* code can safely be executed in parallel, but we still need to decide *where* and *when* parallel execution is beneficial to reduce program execution times. The focus on data-parallel computations and arrays helps here (which is among others why we chose this path in the first place). We do know the index space size of an operation before actually executing it, which is better than in typical divide-and-conquer scenarios.

It is crucial to understand, the WITH-loop does not prescribe parallel execution, it merely opens up opportunities for compiler and runtime system. They still need to make an autonomous decision as whether to make use of this opportunity or not. This sets us apart from many other approaches, may they be as explicit as OpenMP directives or as implicit as `par` and `seq` in Haskell.

Different target architectures require entirely different code generators. In all cases, the SaC compiler does not generate architecture-specific machine code but rather architecture-specific variations of C code. The final step of machine code generation is left to a highly customisable backend compiler tailor-made for a given computing platform. While this design choice foregoes certain machine-level optimisation opportunities, we found it to be a reasonable compromise between engineering effort and support for a variety of computing architectures and operating systems.

The SaC compiler currently supports four different compilation targets. The default target is plain sequential execution. Any ISO/ANSI-compliant C compiler may serve as backend code generator. This flexibility allows us to choose the best performing C compiler on each target architecture, e.g. the Intel compiler for Intel processors, the Oracle compiler for Niagara systems or GNU gcc for AMD Opteron based systems. It would be extremely challenging to compete with these compilers in terms of binary code quality.

For symmetric multi-core multi-processor systems we again target standard ANSI/ISO C with occasional calls to the PThread library. Conceptually, the SaC runtime system follows a fork-join approach, where a program is generally executed by a single *master thread*. Only computationally-intensive kernels, in intermediate SaC code conveniently represented by WITH-loops already enhanced and condensed through high-level optimisation, are effectively run in parallel by temporarily activating a set of a-priori created *worker threads*. The synchronisation and communication mechanisms implementing the transition between single-threaded and multi-threaded execution modes and vice versa are highly optimised to exploit properties of cache coherence protocols found in today's multi-core multi-processor systems. Compilation for these kinds of parallel systems is thoroughly described in [55,56].

As our approach to organising multithreaded execution is not dissimilar from implementations of OpenMP, we recently experimented with alternatively generating C code with OpenMP directives [57]. One result of this work is that (maybe not surprisingly) the tailor-made and highly tuned synchronisation mechanisms of the PThread-based implementation yield slightly better performance. The OpenMP-based code generator may still prove handy for supporting future chip architectures that may not meet our assumptions on cache coherence and memory consistency, but are supported by OpenMP. In either case, PThread- or OpenMP-based code generation, we benefit from the same range of choices to select the most appropriate backend C compiler for binary code generation.

Our support for GPGPUs, the SaC compiler's third target architecture, is based on the CUDA framework [58]. In this case, our design choice to leave binary code generation to an independent C compiler particularly pays off: one is effectively bound to NVidia's custom-made CUDA compiler for code generation.

A number of issues need to be taken into account when targeting graphics cards in general and the CUDA framework in particular that are quite different from generating multithreaded code as before. First CUDA kernels, i.e. the code fragments that actually run on the accelerator, are restricted by the absence of a runtime stack. Consequently, with-loops whose bodies contain function applications that cannot be eliminated by the compiler, e.g. through inlining, disqualify for being run on the graphics hardware. Likewise, there are tight restrictions on the organisation of C-style loop nestings that rule out the transformations for traversing arrays in linear order that are vital on standard multi-core systems. This requires a fairly different path through the compilation process early on. Last but certainly not least, data must be transferred from *host memory* to *device memory* and vice versa before the GPU can participate in any computations, effectively creating a distributed memory. It is crucial for achieving good performance to avoid superfluous memory transfers. The SaC compiler takes all this into account and drastically facilitates the utilisation of many-core graphics accelerators in practice. Details can be found in [59].

The fourth and final target architecture currently supported by the SaC compiler is the MicroGrid architecture [15]. While fairly different from GPGPUs from a computer architecture point of view, it is not dissimilar to CUDA from a code generator perspective. Like CUDA it comes with an architecture-specific programming language embedded into the C language, named μTC, and the corresponding compiler toolchain [60]. The MicroGrid exposes less restrictions on generated C code, but it requires us to expose fine-grained concurrency to the hardware. In essence, the right hand side of Fig. 2 can be seen to illustrate this approach. Whereas in the multithreaded approach the SaC compiler takes considerable effort to adapt the fine-grained concurrency exposed on the program level to the generally much coarser-grained actually available concurrency on the executing hardware platform, the MicroGrid efficiently deals with fine-grained concurrency in hardware. Details can be found in [61,62].

10.6 Experimental Evaluation

To the potential disappointment of our readers space limitations prevent us from any decent analysis as to what extent the SaC compiler achieves its aim of competing with C and Fortran in terms of runtime performance. Instead we refer the interested reader to a number of publications that have exactly this intention. Typically, they put software engineering concerns into context with runtime performance on diverse computing machinery comparing SaC with various other programming languages.

[63] experiments with anisotropic filters and single-class support vector machines from an industrial image processing pipeline. Performance figures are reported from standard commodity multi-core servers and GPGPUs and show competitive performance with respect to hand-coded C implementations and highly customised image processing libraries. [64] investigates scalability issues of the SaC multithreaded runtime system for a number of smaller benchmarks on the Oracle T3-4 server with up to 512 hardware threads. [59] analyses the performance of the GPGPU code generator for a variety of benchmarks.

[65] compares SaC with Fortran-90 in terms of programming productivity and performance on multi-core multi-processor systems for unsteady shock wave interactions. [66] again compares SaC with Fortran-90, this time based on the Kadomtsev-Petiviashvili-I equations (KP-I) that describe the propagation of non-linear waves in a dispersive medium. Last not least, [67] and [68] describe SaC implementations of the NAS benchmarks [69] FT (3-dimensional fast-Fourier transforms) and MG (multigrid), respectively. They show sequential performance for the SaC code that is competitive with the hand-optimised Fortran-77 reference implementations of the two benchmarks and good scalability on multi-processor systems of the pre-multi-core era.

11 Related Work

Given the wide range of topics around the design and implementation of SaC that we have covered in this article, there is a plethora of related work that is impossible to do justice in this section. Hence, the selection inevitably is subjective and incomplete.

General-purpose functional languages such as Haskell, Clean, Sml or OCaml all support arrays in one way or another on the language level. Or more precisely, they support (potentially nested) vectors (1-dimensional arrays) in our terminology. However, as far as implementations are concerned, arrays are rather side issues and design decisions are taken in favour of list- and tree-like data structures. This rules out to achieve competitive performance on array-based compute-intensive kernels.

The most radical step is taken by the ML family of languages: arrays come as stateful, not as functional data structures. To the same degree as this choice facilitates compilation, it looses most appealing characteristics of a functional approach. The lazy functional languages HASKELL and CLEAN both implement fully functional arrays, but investigations have shown that in order to achieve acceptable runtime performance arrays must not only be strict and unboxed (as in SAC), but array processing must also adhere to a stateful regime [70,71,72], i.e. state monads[31] or uniqueness types[33]. While conceptually more elaborate than the ML approach to arrays, monads and uniqueness types likewise enforce an imperative programming style where arrays are explicitly created, copied and removed.

Data Parallel Haskell [73] is an extension of vanilla HASKELL with particular support for nested vectors (arrays in HASKELL speak). Data Parallel Haskell mainly aims at irregular and sparse array problems and inhomogeneous nested vectors in the tradition of NESL[74]. Likewise, it adopts NESL's flattening optimisation that turns nested vectors into flat representations.

One project that must be acknowledged in the context of SAC is SISAL[75,76]. SISAL was the first approach to high-performance functional array programming, and, arguably, it is the only other approach that aims at these goals as stringently as SAC. SISAL predates SAC by about a decade, and consequently, we studied SISAL closely in the early years of the SAC project. Unfortunately, the development of SISAL effectively ended with version 1.1 around the time the first SAC implementation was available. Further developments, such as SISAL 2.0[77] and SISAL-90 [78], were proposed, but have never been implemented.

SAC adopted several ideas of SISAL, e.g. the dispense of many great but implementation-wise costly functional features from currying to higher-order functions and lazy evaluation or non-deferred reference counting to address the aggregate update problem. In many aspects, however, SAC goes far beyond SISAL. Examples are support for truly multi-dimensional arrays instead of 1-dimensional vectors (where only vectors of the same length can be nested in another vector), the ability to define generic abstractions on array operations or the compositional programming style. This list could be extended, but then the comparison is in a sense both unfair and of limited relevance given that development of SISAL ended many years ago.

An interesting offspring from the SISAL project is SAC's namesake SA-C also called Sassy[79,80]. Independently of us and around the same time the originators of SA-C had the idea of a functional language in the spirit of SISAL but with a C-inspired syntax. Thus, we came up with same name: Single Assignment C. Here, the similarities end, even from a syntactic perspective. Despite the almost identical name, SAC and SA-C are very different programming languages.

SAC's implementation of the calculus of multi-dimensional arrays is closely related to interpreted array languages like APL[11,12], J [13] or NIAL[14]. In [81] Bernecky argues that array languages are in principle well suited for data parallel execution and thus should be appropriate for high-performance computing. In practice, language implementations have not followed this path. The main show stopper seems to be the interpretive nature of these languages that hinders code-restructuring optimisations as prominently featured by SaC (Section 10.3). While individual operations could be parallelised, the ratios between productive computation and organisational overhead are often infavourable.

Dynamic (scripting) languages like Python are very popular these days. Consequently, there are serious attempts to establish such languages for compute-intensive applications[82,83]. Here, however, it is very difficult to achieve high performance. Like the Apl-family of languages the highly dynamic nature of programs renders static analysis ineffective. It seems that outside the classical high-performance community, programmers are indeed willing to sacrifice performance in exchange for a more agile software engineering process. Often this is used to explore the design space, and once a proper solution is identified, it is re-implemented with low-level techniques to equip production code with the right performance levels. This is exactly where we see opportunities for SaC: combine agile development with high runtime performance through compilation technology and save the effort of re-implementation and the corresponding consistency issues. Much of the above likewise holds for the arguably most used array language of our time: MatLab and its various clones.

12 Conclusions and Perspectives

We have presented the ins and outs of the programming language Single Assignment C (SaC), covering the whole range of issues from general motivation over language design to programming methodology. In essence, SaC combines array programming technology with functional programming principles and a C-like look-and-feel. In two cases studies on convolution and numerical differentiation we have demonstrated how the SaC methodology supports the engineering of concise, abstract, high-level, reusable code.

However, language design is just one side of the coin. One may even say that this is the easy part. The flip side of the coin is do develop the necessary compiler technology to meet our over-arching objective: competing with the performance of C and Fortran throughout a variety of parallel computing platforms. How to achieve this goal is the real research question behind the SaC project.

An important insight to this end is that before even looking into generating parallel code competitive sequential performance is of paramount importance. Sequential performance is crucial because we aim at exploiting parallel hardware to generate actual performance gains over existing implementations, not to overcome our own shortcomings in sequential performance. While this sounds more than plausible, it truly is a challenge, and a challenge more often avoided than one may think.

Nonetheless, the ability to fully automatically generate code for various parallel architectures, from symmetric multi-core multi-processors to GPGPU accelerators is arguably one of SaC's major assets. In a standard software engineering process the job is less than half done when a first sequential prototype yields correct results. Every targeted parallel architecture requires a different parallelisation approach using different APIs, tools and expertise. Explicit parallelisation is extremely time-consuming and error-prone. Typical programming errors manifest themselves in a non-deterministic way that makes them particularly hard to find. Targeting different kinds of hardware, say multi-core systems and GPGPU-accelerators inevitably clutters the code and creates particular maintenance issues. With SaC the job is done as soon as a sequential program is ready. Multiple parallel target architectures merely require recompilation of the same source code base with different compiler flags.

Much has been achieved since the principal ideas of SaC were first proposed [84]. In Section 10 we sketched out the most important aspects of compilation technology that we have developed to the present day. A series of case studies, further more, A lot of work, nonetheless, lies ahead of us. The continuous development of new parallel architectures keeps us busy just as further improvements of the language and of our compilation infrastructure. Work is currently on-going in many directions, small and large. We conclude this article with sketching out a few of them.

For now, the choice of a target architecture is exclusive. We can either generate code to make use of multiple CPU cores or code to exploit a single GPGPU accelerator. One of our current threads of work is to combine these technologies to make use of multiple GPGPUs and multi-core CPUs at the same time. This work also accounts for current hardware trends to combine CPU and GPU technologies on-chip.

Another area of on-going work is to exploit the capabilities of vector registers and vector operations available in most of today's processors. The most prominent example are Intel's Streaming SIMD Extensions (SSE) for the x86 architecture, but similar features are included in all modern processor designs. At the moment, SaC does not explicitly exploits these facilities and leaves their potential to be exploited by the backend compiler. Since the SaC compiler has a much better understanding of its intermediate code than any backend C compiler could ever derive from the generated code, it would be desirable to generate vector instructions explicitly in `sac2c`. Unfortunately, the multitude of ISA extensions and APIs is rather cumbersome. We also need to extend the set of SaC base types to take full advantage of vector registers. Currently, we explore ways to support user-defined bit widths for numerical values.

Fig. 3 in the very beginning of this article already outlined two directions of on-going work. As of now, SaC does not support network-interconnected clusters or, generally, distributed memory architectures. Despite the multicore revolution, it is always attractive to combine multiple complete systems for even larger computational tasks. On the other end of the design space we envision a growing relevance of reconfigurable hardware to address tomorrow's demands on energy efficiency. One can even think of reconfigurable areas in general-purpose processors. Right now, programming reconfigurable hardware requires a completely different tool and mind set than conventional software engineering. However, SaC intermediate code appears to be a suitable starting point for compilation.

Our namesake SA-C/Sassy (see Section 11) took a similar approach about a decade ago, but was presumably ahead of its time. It is fair to say that our efforts into both directions are still in their infancy.

On the language level a number of features are highly desirable. As the participants of the CEFP summer school (painfully) learned during the lab sessions, the monomorphic type system for array base types is suboptimal. While it is easy to define new types in SaC, dealing with arrays of user-defined types is less easy. All support for our advocated compositional programming methodology is based on shape-generic but base-type-monomorphic function definitions in the SaC standard library. These are, of course, not available to (arrays of) user-defined types and need to be provided for each such type by its originator. Polymorphism on base types would immediately solve this issue, but realisation in the context of shapely polymorphism, overloading and the strong desire not to loose on the performance side of the coin create a challenging research question that is currently under investigation.

Another type system issue has been discussed already: the SaC array type system does not support the specification of relationships of the shapes of function arguments and results. For example, matrix multiplication can only be specified for 2-dimensional arrays of any shape, whereas the algorithm requires the y-axis extent of the first argument to coincide with the x-axis extent of the second argument. Furthermore, the algorithm reveals that the result matrix has the same size along the x-axis as the first argument matrix and the same size on the y-axis as the second argument matrix. This knowledge is lost in the type system due to a lack of expressiveness. Similar shape relations are common place across SaC standard array operations, e.g. `take`, `drop` or `where`. Capturing such shape relations in the type system leads to dependent array types that we have studied in the context of (more experimental) array language Qube [28,29]. However, how to carry these ideas over to SaC in the presence of overloading and dynamic dispatch is non-trivial.

Of course, as functional programmers we have a longer wish list for the feature set of SaC. While SaC will always put the emphasis on arrays, it would be highly desirable to support tuples, lists and trees, nonetheless. Likewise, higher-order functions are certainly worthwhile some implementation effort. Currently, the `fold`-WITH-loop are the only place where functions appear in an (almost) expression position, but this is very restricted and does not allow for abstractions like a general reduction function or operator. Not adopting the general concept of higher-order functions was an early design decisions to facilitate compilation into efficient code. However, restricted support for higher-order functions such that the compiler could in practice resolve them may nonetheless bring a considerable gain in expressiveness.

Acknowledgements. The work described in this paper is the result of more than 15 years of research and development conducted by an ever changing group of people working at a variety of places. From the nucleus at the University of Kiel, Germany, in the mid-1990s the virus spread to the University of Lübeck, Germany, the University of Hertfordshire, England, the University of Toronto, Canada, the University of Amsterdam, Netherlands, and recently to Heriot-Watt University, Scotland. Apart from the internal funds of these universities, three European projects have been instrumental in supporting our activities: Æther, Apple-CORE and Advance.

First and foremost, I would like to thank Sven-Bodo Scholz for many years of intense and fruitful collaboration. The original proposal of a no-frills functional language with a C-like syntax and particular support for arrays was his [84]. Apart from the name and these three design principles not too much in today's SaC resembles the original proposal, though.

My special thanks go to those who helped to shape SaC by years of continued work: Dietmar Kreye, Robert Bernecky, Stephan Herhut and Kai Trojahner. Over the years many more have contributed to advancing SaC to its current state. I take the opportunity to thank (in roughly temporal order) Henning Wolf, Arne Sievers, Sören Schwartz, Björn Schierau, Helge Ernst, Jan-Hendrik Schöler, Nico Marcussen-Wulff, Markus Bradtke, Borg Enders, Michael Werner, Karsten Hinckfuß, Steffen Kuthe, Florian Massel, Andreas Gudian, Jan-Henrik Baumgarten, Theo van Klaveren, Daoen Pan, Sonia Chouaieb, Florian Büther, Torben Gerhards, Carl Joslin, Jing Guo, Hraban Luyat, Abhishek Lal, Artem Shinkarov, Santanu Dash, Daniel Rolls, Zheng Zhangzheng, Aram Visser, Tim van Deurzen, Roeland Douma, Fangyong Tang, Pablo Rauzy and Miguel Diogo for their invaluable work.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38 (1965)
2. Sutter, H.: The free lunch is over: A fundamental turn towards concurrency in software. Dr. Dobb's Journal 30 (2005)
3. Meuer, H., Strohmaier, E., Simon, H., Dongarra, J.: 38th top500 list (2011), `www.top500.org`
4. Intel: Product Brief: Intel Xeon Processor 7500 Series. Intel (2010)
5. AMD: AMD Opteron 6000 Series Platform Quick Reference Guide. AMD (2011)
6. Koufaty, D., Marr, D.: Hyperthreading technology in the netburst microarchitecture. IEEE Micro 23, 56–65 (2003)
7. Sun/Oracle: Oracle's SPARC T3-1, SPARC T3-2, SPARC T3-4 and SPARC T3-1B Server Architecture. Whitepaper, Oracle (2011)
8. Shin, J.L., Huang, D., Petrick, B., et al.: A 40 nm 16-core 128-thread SPARC SoC processor. IEEE Journal of Solid-State Circuits 46, 131–144 (2011)
9. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. Int. Journal of Parallel Programming 34, 383–427 (2006)
10. Grelck, C., Scholz, S.B.: SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In: Glew, N., Blelloch, G. (eds.) 2nd Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), Nice, France, pp. 25–33. ACM Press (2007)
11. Falkoff, A., Iverson, K.: The Design of APL. IBM Journal of Research and Development 17, 324–334 (1973)
12. International Standards Organization: Programming Language APL, Extended. ISO N93.03, ISO (1993)

13. Hui, R.: An Implementation of J. Iverson Software Inc., Toronto (1992)
14. Jenkins, M.: Q'Nial: A Portable Interpreter for the Nested Interactive Array Language Nial. Software Practice and Experience 19, 111–126 (1989)
15. Bousias, K., Guang, L., Jesshope, C., Lankamp, M.: Implementation and Evaluation of a Microthread Architecture. J. Systems Architecture 55, 149–161 (2009)
16. Schildt, H.: American National Standards Institute, International Organization for Standardization, International Electrotechnical Commission, ISO/IEC JTC 1: The annotated ANSI C standard: American National Standard for Programming Languages C: ANSI/ISO 9899-1990. McGraw-Hill (1990)
17. Kernighan, B., Ritchie, D.: The C Programming Language. Prentice-Hall (1988)
18. Iverson, K.: A Programming Language. John Wiley (1962)
19. Iverson, K.: Programming in J. Iverson Software Inc., Toronto (1991)
20. Burke, C.: J and APL. Iverson Software Inc., Toronto (1996)
21. Jenkins, M., Jenkins, W.: The Q'Nial Language and Reference Manual. Nial Systems Ltd., Ottawa (1993)
22. Mullin, L.R., Jenkins, M.: A Comparison of Array Theory and a Mathematics of Arrays. In: Arrays, Functional Languages and Parallel Systems, pp. 237–269. Kluwer Academic Publishers (1991)
23. Mullin, L.R., Jenkins, M.: Effective Data Parallel Computation using the Psi Calculus. Concurrency — Practice and Experience 8, 499–515 (1996)
24. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Transactions on Computational Science and Engineering 5 (1998)
25. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press (2008)
26. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press (1994)
27. Douma, R.: Nested Arrays in Single Assignment C. Master's thesis, University of Amsterdam, Amsterdam, Netherlands (2011)
28. Trojahner, K., Grelck, C.: Dependently Typed Array Programs Don't Go Wrong. Journal of Logic and Algebraic Programming 78, 643–664 (2009)
29. Trojahner, K.: QUBE — Array Programming with Dependent Types. PhD thesis, University of Lübeck, Lübeck, Germany (2011)
30. Grelck, C., Scholz, S.B.: Axis Control in SAC. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 182–198. Springer, Heidelberg (2003)
31. Wadler, P.: Comprehending Monads. Mathematical Structures in Computer Science 2 (1992)
32. Peyton Jones, S., Launchbury, J.: State in Haskell. Lisp and Symbolic Computation 8, 293–341 (1995)
33. Smetsers, S., Barendsen, E., van Eekelen, M., Plasmeijer, M.: Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. Technical report, University of Nijmegen, Nijmegen, Netherlands (1993)
34. Achten, P., Plasmeijer, M.: The ins and outs of Clean I/O. Journal of Functional Programming 5, 81–110 (1995)
35. Grelck, C.: Integration eines Modul- und Klassen-Konzeptes in die funktionale Programmiersprache SAC – Single Assignment C. Master's thesis, University of Kiel, Germany (1996)
36. Grelck, C., Scholz, S.B.: Classes and Objects as Basis for I/O in SAC. In: 7th International Workshop on Implementation of Functional Languages (IFL 1995), Båstad, Sweden, pp. 30–44. Chalmers University of Technology, Gothenburg (1995)

37. Herhut, S., Scholz, S.B., Grelck, C.: Controllling Chaos — On Safe Side-Effects in Data-Parallel Operations. In: 4th Workshop on Declarative Aspects of Multicore Programming (DAMP 2009), Savannah, USA, pp. 59–67. ACM Press (2009)
38. Grelck, C., Scholz, S., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming 38, 38–67 (2010)
39. Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13, 1005–1059 (2003)
40. Grelck, C., van Deurzen, T., Herhut, S., Scholz, S.B.: Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. Concurrency and Computation: Practice and Experience (2011)
41. Scholz, S.-B.: WITH-Loop-Folding in SAC - Condensing Consecutive Array Operations. In: Clack, C., Hammond, K., Davie, T. (eds.) IFL 1997. LNCS, vol. 1467, pp. 72–92. Springer, Heidelberg (1998)
42. Grelck, C., Hinckfuß, K., Scholz, S.B.: With-Loop Fusion for Data Locality and Parallelism. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 178–195. Springer, Heidelberg (2006)
43. Grelck, C., Scholz, S.-B., Trojahner, K.: With-Loop Scalarization – Merging Nested Array Operations. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 118–134. Springer, Heidelberg (2004)
44. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing 32, 507–522 (2006)
45. Bernecky, R., Herhut, S., Scholz, S.-B., Trojahner, K., Grelck, C., Shafarenko, A.: Index Vector Elimination – Making Index Vectors Affordable. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 19–36. Springer, Heidelberg (2007)
46. Grelck, C.: Improving Cache Effectiveness through Array Data Layout Manipulation in SAC. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 231–248. Springer, Heidelberg (2001)
47. Grelck, C., Kreye, D., Scholz, S.B.: On Code Generation for Multi-Generator WITH-Loops in SAC. In: Koopman, P., Clack, C. (eds.) IFL 1999. LNCS, vol. 1868, pp. 77–94. Springer, Heidelberg (2000)
48. Wilson, P.R.: Uniprocessor Garbage Collection Techniques. In: Bekkers, Y., Cohen, J. (eds.) IWMM 1992. LNCS, vol. 637, pp. 1–42. Springer, Heidelberg (1992)
49. Jones, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley (1999)
50. Marlow, S., Harris, T., James, R.P., Peyton Jones, S.: Parallel generational-copying garbage collection with a block-structured heap. In: 7th Int. Symposium on Memory Management (ISMM 2008), Tucson, AZ, USA, pp. 11–20. ACM (2008)
51. Hudak, P., Bloss, A.: The Aggregate Update Problem in Functional Programming Systems. In: 12th ACM Symposium on Principles of Programming Languages (POPL 1985), New Orleans, USA, pp. 300–313. ACM Press (1985)
52. Collins, G.E.: A Method for Overlapping and Erasure of Lists. CACM 3, 655–657 (1960)
53. Grelck, C., Trojahner, K.: Implicit Memory Management for SaC. In: 16th International Workshop on Implementation and Application of Functional Languages, IFL 2004, Lübeck, Germany, pp. 335–348. University of Kiel, Institute of Computer Science and Applied Mathematics (2004); Technical Report 0408

54. Grelck, C., Scholz, S.B.: Efficient Heap Management for Declarative Data Parallel Programming on Multicores. In: 3rd Workshop on Declarative Aspects of Multicore Programming (DAMP 2008), San Francisco, CA, USA, pp. 17–31. ACM Press (2008)
55. Grelck, C.: A Multithreaded Compiler Backend for High-Level Array Programming. In: 2nd International Conference on Parallel and Distributed Computing and Networks (PDCN 2003), Innsbruck, Austria, pp. 478–484. ACTA Press (2003)
56. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming 15, 353–401 (2005)
57. Zhangzheng, Z.: Using OpenMP as an Alternative Parallelization Strategy in SAC. Master's thesis, University of Amsterdam, Amsterdam, Netherlands (2011)
58. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann (2010)
59. Guo, J., Thiyagalingam, J., Scholz, S.B.: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: 6th Workshop on Declarative Aspects of Multicore Programming (DAMP 2011), Austin, TX, USA. ACM Press (2011)
60. Bernard, T., Grelck, C., Jesshope, C.: On the compilation of a language for general concurrent target architectures. Parallel Processing Letters 20, 51–69 (2010)
61. Herhut, S., Joslin, C., Scholz, S.B., Grelck, C.: Truly Nested Data-Parallelism: Compiling SAC to the Microgrid Architecture. In: 21st Symposium on Implementation and Application of Functional Languages (IFL 2009), South Orange, NJ, USA. Seton Hall University (2009)
62. Herhut, S., Joslin, C., Scholz, S.-B., Poss, R., Grelck, C.: Concurrent Non-deferred Reference Counting on the Microgrid: First Experiences. In: Hage, J., Morazán, M.T. (eds.) IFL 2010. LNCS, vol. 6647, pp. 185–202. Springer, Heidelberg (2011)
63. Wieser, V., Grelck, C., Haslinger, P., Guo, J., Korzeniowski, F., Bernecky, R., Moser, B., Scholz, S.: Combining high productivity and high performance in image processing using Single Assignment C on multi-core cpus and many-core gpus. Journal of Electronic Imaging (to appear)
64. Grelck, C., Douma, R.: SAC on a Niagara T3-4 Server: Lessons and Experiences. In: 15th Int. Conference on Parallel Computing (ParCo 2011), Ghent, Belgium (2011)
65. Rolls, D., Joslin, C., Kudryavtsev, A., Scholz, S.-B., Shafarenko, A.: Numerical Simulations of Unsteady Shock Wave Interactions Using SaC and Fortran-90. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 445–456. Springer, Heidelberg (2009)
66. Shafarenko, A., Scholz, S.B., Herhut, S., Grelck, C., Trojahner, K.: Implementing a Numerical Solution of the KPI Equation using Single Assignment C: Lessons and Experiences. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 160–177. Springer, Heidelberg (2006)
67. Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In: Malyshkin, V.E. (ed.) PaCT 2003. LNCS, vol. 2763, pp. 230–235. Springer, Heidelberg (2003)
68. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In: Prasanna, V.K., Westrom, G. (eds.) 16th International Parallel and Distributed Processing Symposium (IPDPS 2002), Fort Lauderdale, USA. IEEE Computer Society Press (2002)
69. Bailey, D., et al.: The NAS Parallel Benchmarks. International Journal of Supercomputer Applications 5, 63–73 (1991)
70. van Groningen, J.: The Implementation and Efficiency of Arrays in Clean 1.1. In: Kluge, W.E. (ed.) IFL 1996. LNCS, vol. 1268, pp. 105–124. Springer, Heidelberg (1997)

71. Zörner, T.: Numerical Analysis and Functional Programming. In: 10th International Workshop on Implementation of Functional Languages (IFL 1998), London, UK, University College, pp. 27–48 (1998)
72. Chakravarty, M.M., Keller, G.: An Approach to Fast Arrays in Haskell. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 27–58. Springer, Heidelberg (2003)
73. Peyton Jones, S., Leshchinskiy, R., Keller, G., Chakravarty, M.: Harnessing the multicores: Nested data parallelism in Haskell. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008), Bangalore, India, pp. 383–414 (2008)
74. Blelloch, G., Chatterjee, S., Hardwick, J., Sipelstein, J., Zagha, M.: Implementation of a Portable Nested Data-Parallel Language. Journal of Parallel and Distributed Computing 21, 4–14 (1994)
75. McGraw, J., Skedzielewski, S., Allan, S., Oldehoeft, R., et al.: Sisal: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. M 146. Lawrence Livermore National Laboratory, Livermore (1985)
76. Cann, D.: Retire Fortran? A Debate Rekindled. CACM 35, 81–89 (1992)
77. Oldehoeft, R.: Implementing Arrays in SISAL 2.0. In: 2nd SISAL Users Conference, San Diego, CA, USA, pp. 209–222. Lawrence Livermore National Laboratory (1992)
78. Feo, J., Miller, P., Skedzielewski, S.K., Denton, S., Solomon, C.: Sisal 90. In: Conference on High Performance Functional Computing (HPFC 1995), Denver, CO, USA, pp. 35–47. Lawrence Livermore National Laboratory, Livermore (1995)
79. Hammes, J., Draper, B., Böhm, A.: Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems. In: Christensen, H.I. (ed.) ICVS 1999. LNCS, vol. 1542, pp. 83–97. Springer, Heidelberg (1999)
80. Najjar, W., Böhm, W., Draper, B., Hammes, J., et al.: High-level Language Abstraction for Reconfigurable Computing. IEEE Computer 36, 63–69 (2003)
81. Bernecky, R.: The Role of APL and J in High-Performance Computation. APL Quote Quad. 24, 17–32 (1993)
82. van der Walt, S., Colbert, S., Varoquaux, G.: The numpy array: A structure for efficient numerical computation. Computing in Science & Engineering 13 (2011)
83. Kristensen, M., Vinter, B.: Numerical Python for scalable architectures. In: 4th Conference on Partitioned Global Address Space Programming Model (PGAS 2010). ACM Press, New York (2010)
84. Scholz, S.B.: Single Assignment C – Functional Programming Using Imperative Style. In: 6th International Workshop on Implementation of Functional Languages (IFL 1994), pp. 21.1–21.13. University of East Anglia, Norwich (1994)