SaC 1.0 — Single Assignment C — Tutorial

Scholz, S.-B.; Herhut, S.; Penczek, F.; Grelck, C.

Publication date
2010
Document Version
Submitted manuscript

University of Hertfordshire
School of Computer Science

University of Amsterdam
Institute of Informatics

SaC 1.0

— Single Assignment C —

Tutorial

Sven-Bodo Scholz, Stephan Herhut,
Frank Penczek, Clemens Grelck

December 2010

Contents

Part I

Trails Covering the Basics of SaC

Chapter 1

Running the first program

The following instructions will help you write your first SaC program.

1.1 A Checklist

To successfully write and run your first SaC program, you will need:

- An **ANSI C compiler**, such as `gcc`. Though not needed directly, the SaC compiler relies on it.
- The **SaC compiler** `sac2c`. It can be downloaded at <http://www.sac-home.org/>. For convenience, you should make sure, that `sac2c` is located in your `$PATH`.
- The **SaC standard library** that comes with `sac2c` as part of the SaC distribution. To make sure `sac2c` will be able to find it, the environment variable `$SACBASE` should point to it, i.e., `ls $SACBASE` should yield at least the subdirectory `stdlib`.
- Your favorite **text editor**, such as `vi` or `emacs`.

1.2 Create your first SaC Source File

Start your editor and type the following program:

Listing 1.1: Hello World

```
1 use StdIO: all;
2 use Array: all;
3
4 int main()
5 {
6   printf( "Hello World!\n");
7   return(0);
8 }
```

As you can see, it has a strong resemblance to C. The major difference are the module use declarations at the beginning of the program. For now, it suffices to keep in mind, that these two use declarations for most experiments will do the job.

In order to proceed, save this program into a file named `world.sac`.

1.3 Compile the Source File and Run the Program

The SaC compiler invocation is similar to the standard invocation of C compilers. A typical shell session for compiling `world.sac` could be:

```
> cd /home/sbs/sac/
> ls
world.sac
> sac2c world.sac
> ls
a.out      a.out.c    world.sac
> a.out
Hello World!
>
```

The compilation process consists of two steps. First, the SaC compiler generates a C file, which then is compiled into target code by utilizing the system's C compiler. If no target file name is specified, the intermediate C file is named `a.out.c` so that the subsequent invocation of the C compiler creates an executable named `a.out`.

In the same way the default target name `a.out` is borrowed from standard C compilers, the `-o` option for specifying target names is adopted as well. For example, `sac2c -o world world.sac` results in files `world.c` and `world`.

Note here, that the compiled program, depending on the operating system, is linked either statically or dynamically. However, it does not require any further linking or interpretation.

Chapter 2

Array Programming Basics

This trail gives an introduction to the basic concepts of array programming in SaC. It consists of two lessons: **Arrays as Data** and **Shape-Invariant Programming**. In the former lesson, the major differences between arrays in SaC and arrays in more mainstream languages are explained. The lesson **Shape-Invariant Programming** gives an introduction into the most important array operations available in SaC. Based on these operations, several small examples demonstrate how more complex array operations can be constructed by simply combining the basic ones.

2.1 Lesson: Arrays as Data

In SaC, arrays are the only data structures available. Even scalar values are considered arrays. Each array is represented by two vectors, a so-called **shape vector** and a **data vector**. An array's shape vector defines its **shape**, i.e., its extent within each axis, and its **dimensionality** (or **rank**), which is given implicitly by the shape vector's length.

The section on **Defining Arrays** explains how arrays of various dimensionality can be defined in SaC, and how they can be generated via nesting. Furthermore, some elementary notation such as **scalars**, **vectors**, and **matrices** is defined.

The section on **Arrays and Variables** discusses the purely functional array model used in SaC.

2.1.1 Defining Arrays

In this section, several means for specifying arrays are explained.

In principle, all arrays in SaC can be defined by using the `reshape` operation. `reshape` expects two operands, a shape vector and a data vector, both of which are specified as comma separated lists of numbers enclosed in square brackets.

To get started, try the following program:

Listing 2.1: Defining Arrays I

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    print( reshape( [5], [1,2,3,4,5]));
7    print( reshape( [3,2], [1,2,3,4,5,6]));
8    print( reshape( [3,2,1], [1,2,3,4,5,6]));
9    return(0);
10 }
```

It prints three arrays:

- an array of dimensionality 1 with 5 elements `[1,2,3,4,5]`;
- an array of dimensionality 2 with 3 rows and 2 columns, and
- a 3-dimensional array with 3 elements in the leftmost axis, 2 elements in the middle axis, and one element in the rightmost axis.

Note here, that the function **print** can be applied to arbitrary arrays. Besides printing its argument's dimensionality and shape, i.e., its shape vector, a more intuitive representation of the array's data vector is shown. However, as the terminal allows for 2 dimensions only, arrays of higher dimensionality are interpreted as nestings of 2-dimensional arrays. Therefore, the 3-dimensional array is printed as a 2-dimensional array of vectors.

Exercise 1 *In all these examples, the product of the shape vector matches the length of the data vector. What do you expect to happen, if this condition does not hold?*

For reasons of convenience, we use the following terminology:

scalar always denotes an array of dimensionality 0,

vector always denotes an array of dimensionality 1, and

matrix always denotes an array of dimensionality 2.

As **all** arrays can be defined in terms of `reshape`, the following program as well is perfectly legal:

Listing 2.2: Defining Arrays II

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    print( reshape( [1], [1]));
7    print( reshape( [], [1]));
8    return(0);
9  }
```

The most interesting aspect of this program is the array defined in line 7. The empty shape vector makes it a 0-dimensional array, i.e., a scalar. The data vector carries the scalar's value, which, in this example, is 1.

Exercise 2 *The arguments of* `reshape` *are vectors, i.e., arrays of dimensionality 1. Can they be specified by* `reshape` *expressions themselves?*

The `reshape` notation is relatively clumsy, in particular, when being used for scalars. Therefore, scalars and vectors can alternatively be specified by shortcut notations as well.

For experimenting with these, try the following:

Listing 2.3: Shortcut Notation for Arrays

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    print( 1);
7    print( [1,2,3,4,5]);
8    print( [ [1,2], [3,4], [5,6]]);
9    print( genarray( [4,3,2], 1));
10   print( genarray( [4,3], [1,2]));
11   return(0);
12 }
```

From these examples, we can see that scalars can be used in the same way as in most programming languages, and that the notation used for the parameters of `reshape` in the examples above in fact is a standard abbreviation of SaC. The example in line 8 shows that nestings of arrays are implicitly eliminated, i.e., the resulting array is identical to

`reshape( [3,2], [1,2,3,4,5,6]).`

For this reason, array nestings in SaC always have to be **homogeneous**, i.e., the inner array components have to have identical shapes.

Furthermore, a new function is introduced: `genarray`. It expects two arguments, a shape vector that defines the shape of the result and a default element to be inserted at each position of the result. As shown in the example of line 10, the individual array elements can be non-scalar arrays as well, which implicitly extends the dimensionality of the result array.

Exercise 3 *Given the language constructs introduced so far, can you define an array that would print as*

```
Dimension:  3
Shape    : <  5,  2,  2>
< 0  0 > < 0  0 >
< 1  0 > < 0  0 >
< 0  1 > < 0  0 >
< 0  0 > < 1  0 >
< 0  0 > < 0  1 >
```

, but whose definition does not contain the letter `1` *more than once?*

2.1.2 Arrays and Variables

This section explains why in SaC arrays are data and not containers for values as found in most other languages.

So far, all examples were expression based, i.e., we did not use any variables. Traditionally, there are two different ways of introducing variables. In conventional (imperative) languages such as C, variables denote memory locations which hold values that may change during computation. In functional languages, similar to mathematics, variables are considered place holders for values. As a consequence, a variable's value can never change. Although this difference may seem rather subtle at first glance, it has quite some effects when operations on large data structures (in our case: large arrays) are involved.

Let's have a look at an example:

Listing 2.4: Variables as Placeholders

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = [1,2,3,4];
7    print( a);
8
9    b = modarray( a, [0], 9);
10   print( b);
11
12   return(0);
13 }
```

The function `modarray` expects three arguments: an array to be "modified", an index that indicates the exact position within the array to be "modified", and the value that is to be inserted at the specified position. As we would expect, the resulting array `b` is almost identical to `a`, only the very first element has changed into `9`.

Note here, that indexing in SaC always starts with index `0`!

Referring to the container / place holder discussion, the crucial question is: does the variable `a` denote a container, whose value is changed by `modarray`? If this would be the case, `a` and `b` would share the same container, and every access to `a` after line 9 would yield `[9,2,3,4]`. If `a` in fact is a place holder, it will always denote the array `[1,2,3,4]`, no matter what functions have obtained `a` as an argument.

To answer this question, you may simply shift the first call of `print` two lines down. As you can see, in SaC, variables are indeed place holders.

> **A note for efficiency freaks:**
> *You may wonder whether this implies that* `modarray` *always copies the entire array. In fact, it only copies* `a` *if the place-holder property would be violated otherwise.*

As a result of this place-holder property, it is guaranteed that no function call can affect the value of its arguments. In other words, the underlying concept

guarantees, that all functions are "pure". Although this helps in avoiding nasty errors due to non-intended side-effects, it sometimes seems to be an annoyance to always invent new variable names, in particular, if arrays are to be modified successively.

To cope with this problem, in SaC, variables do have a so-called **scope**, i.e., each variable definition is associated to a well-defined portion of program code where its definition is valid. In a sequence of variable definitions, the scope of a variable starts with the left-hand side of its definition and either reaches down to the end of the function, or, provided at least one further definition of a variable with the same name exists, to the right-hand side of the next such definition. This measure allows us to reuse variable names. A slight modification of our example demonstrates the effect of variable scopes in SaC:

Listing 2.5: Variable Scopes

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = [1,2,3,4];
7
8    b = modarray( a, [0], 9);
9    print( a);
10   a = b;
11   print( a);
12
13   a = modarray( a, [1], 8);
14   print(a);
15
16   return(0);
17 }
```

Here, the use of `a` on the right-hand side of line 9 still refers to the definition of line 6, whereas the use in line 11 refers to the definition in line 10.

The definition in line 13 shows, how variable scopes can be used to specify code that looks very much "imperative". However, you should always keep in mind, that in SaC, the place-holder property **always** holds!

Exercise 4 *What result do you expect from the following SaC program?*

Listing 2.6: Scope Exercise

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = [1,2,3,4];
7    b = [a,a];
8
9    a = modarray( modarray( a, [0], 0), [1], 0);
10   b = modarray( b, [0], a);
11   print(b);
12
13   return(0);
14 }
```

2.2 Lesson: Shape-Invariant Programming

The term **shape-invariant programming** refers to a programming style where all array operations are defined in terms of operations that are applied to entire arrays rather than to individual array elements. In order to realize such a programming style, it is an essential prerequisite to be able to apply functions to arbitrarily shaped arguments. In SaC, this is the case.

All built-in array operations as well as all array operations supplied by the standard library can be applied to arguments of arbitrary shapes. However, most of the operations that require more than one argument do have certain restrictions with respect to which array shapes can be combined as valid arguments. If an operation is applied to a set of arguments whose shapes constitute an illegal combination, usually, a type error message is given. In cases where this cannot be decided statically, the compiler may accept such a kind of domain error and produce a runtime error instead.

This lesson consists of three parts: The section on **Standard Array Operations** introduces the most important standard array operations provided by the current SaC compiler release [1]. The next section explains **Axis Control Notation**, a powerful but simple way of manipulating the focus of array operations with respect to individual axes of argument arrays. With the axis-control notation, the basic operations often can easily be combined into rather complex operations as demonstrated in the section on **Putting it all Together**.

2.2.1 Standard Array Operations

In the sequel, several toy examples demonstrate the functionality of the most basic array operations that come as part of the current SaC release. Their design is inspired by those available in APL. However, several aspects - in particular regarding the treatment of special cases in APL - have been adjusted to allow for a more favourable compilation in SaC that yields better runtime performance.

> **A note for language design freaks:**
> *You may have your own ideas on what primitive array operations should be available and how the precise semantics of these should look like. Therefore, it should be mentioned here, that* **all(!)** *array operations introduced in the remainder of this section are not hard-wired into the language, but they are defined in the module* `Array` *from the standard library. This is to say that the advanced SaC programmer may write his own set of standard array operations.*

The individual parts of this section are all organized according to the following scheme: first, a semi-formal introduction to the functionality of individual operations is given. Then, several examples shed some more light on the exact semantics of each operation by varying the argument-shape constellations and

[1] As of this writing, the latest SaC compiler release is version 1.00-beta.

by exploring "border-line cases" with respect to domain restrictions if these do exist.

Basic Operations

The most basic operations are very close to the model of arrays in SaC. They comprise functions for inspecting, creating, and modifying an array's shape and content. If not stated otherwise, they are applicable to arbitrarily shaped arguments of built-in element type. Note here, that some of them have been introduced in earlier lessons already.

dim(a) returns the (scalar) dimensionality of the argument array `a`.
Domain restrictions: none.

shape(a) returns the shape vector of the argument array `a`.
Domain restrictions: none.

a[iv] constitutes a short-cut notation for `sel( iv, a)`. It selects the array element of `a` at index position `iv`. As `a` may be of any shape, the index position is given as an **index vector**. The dimensionality of the result is identical to the dimensionality of `a` minus the length of `iv`. Accordingly, its shape is derived from the last components of the shape of `a`.
Domain restrictions:

- `dim(iv)` $=$ `1`
- `shape(iv)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ..., $`shape(iv)[[0]]`$\}$: `iv[[`$i$`]]` $<$ `shape(a)[[`i`]]`.

a[iv]=expr; is a short-cut notation for an assignment of the form `a = modarray( a, iv, expr);`. The result of this application is a new array which is almost identical to `a`. Only the element (subarray) at index position `iv` is different; it is replaced by `expr`.
Domain restrictions:

- `dim(iv)` $=$ `1`
- `shape(iv)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ..., $`shape(iv)[[0]]`$\}$: `iv[[`$i$`]]` $<$ `shape(a)[[`i`]]`
- `shape(expr)` $=$ `shape(a[iv])`.

reshape(shp, expr) computes an array with shape vector *shp* and data vector identical to that of *expr*.
Domain restrictions:

- `dim( shp)` $=$ `1`
- $\prod_{i=0}^{\texttt{shape(shp)[[0]]}-1}$ `shp[[`i`]]` $=$ $\prod_{i=0}^{\texttt{dim(expr)}-1}$ `shape(expr)[[`i`]]`.

genarray(shp, expr) generates an array of shape `shp`, whose elements are all identical to `expr`.
Domain restrictions: `dim(shp) = 1`.

Although these operations are fairly self-explaining or known from Lesson 2.1 on **Arrays as Data**, let us have a look at a few example applications:

Listing 2.7: Basic Operations

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9,10,11,12];
7
8    mat = reshape( [3,4], vect);
9    print( mat);
10
11   print( mat[[1,1]]);
12   print( mat[[2]]);
13   print( mat[[]]);
14
15   mat[[1,1]] = 0;
16   print( mat);
17   mat[[2]] = [0,0,0,0];
18   print( mat);
19   mat[[]] = genarray( [3,4], 0);
20   print( mat);
21
22   empty_vect = [];
23   print( empty_vect);
24   empty_mat = reshape( [22,0], empty_vect);
25   print( empty_mat);
26   print( dim( empty_mat));
27   print( shape( empty_mat));
28
29   return(0);
30 }
```

The different selections in Lines 11-13 show how the dimensionality of the selected element increases as the length of the index vector decreases. If the index vector degenerates into an empty vector, the entire array is selected. Similarly, the applications of modarray in Lines 15-20 demonstrate the successive replacement of individual elements, rows, or the entire array.

Lines 22-27 are meant to draw the reader's attention to the fact that there exists an unlimited number of distinct empty arrays in SaC!

Exercise 5 *Assuming* `mat` *to be defined as in the previous example, what results do you expect from the following expressions:*

- `reshape([3,0,5], [])[[]]`*?*
- `reshape([3,0,5], [])[[1]]`*?*
- `reshape([3,0,5], [])[[1,0]]`*?*
- `mat[reshape([2,0], [])]`*?*

Element-wise Extensions

Most of the operations that can be found as standard operations on scalars in other languages are applicable to entire arrays in SaC. Their semantics are simply element-wise extensions of the well-known scalar operations. The binary operations in general do have some domain restrictions. First, the element types of both arguments do have to be identical. Furthermore, either one of the arguments has to be a scalar value, or both arguments have to have identical shapes. In the former case, the value of the scalar argument is combined with each element of the (potentially non-scalar) other argument, which dictates the shape of the result. The latter case results in an array of the same shape as both arguments are, with the values being computed from the corresponding elements of the argument arrays.

In detail, the following operations are available:

arithmetic operations including addition (`e1 + e2`), subtraction (`e1 - e2`), negation (`- e1`), multiplication (`e1 * e2`), and division (`e1 / e2`). Furthermore, division on rational numbers (`e1 div e2`) and a modulo operation (`e1 % e2`) are supported.
Domain restrictions: the element types have to be numerical.

logical operations including conjunction (`e1 && e2`), disjunction (`e1 || e2`), and negation (`! e1`).
Domain restrictions: the element types have to be Boolean.

relational operations including less-than (`e1 < e2`), less-or-equal (`e1 <= e2`), equal (`e1 == e2`), not-equal (`e1 != e2`), greater-or-equal (`e1 >= e2`), and greater-than (`e1 > e2`).
Domain restrictions: none.

max(e1, e2), min(e1, e2) compute the element-wise maximum and minimum, respectively.
Domain restrictions: none.

where(p, e1, e2) is an element-wise extension of a conditional. It expects `p`, `e1`, and `e2` either to have identical shapes or to be scalar. If at least one of the three arrays is non-scalar, that shape serves as shape of the result, whose values are taken from `e1` or `e2` depending on the value(s) of `p`.
Domain restrictions:

- the element type of `p` has to be boolean
- the element types of `e1` and `e2` have to be identical
- $\exists$ `shp`: ((`shape(p)` = `shp` $\vee$ `shape(p)` = `[]`) $\wedge$ (`shape(e1)` = `shp` $\vee$ `shape(e1)` = `[]`) $\wedge$ (`shape(e2)` = `shp` $\vee$ `shape(e2)` = `[]`)).

Again, these operations are fairly self-explanatory. Nevertheless, we present a few examples:

Listing 2.8: Elementwise Extensions

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9];
7
8    mat = [ vect, vect+10, vect+20];
9    print( mat);
10
11   mat2 = where( (mat % 2) == 0, mat, -mat);
12   print(mat2);
13
14   print( max( mat2, 0));
15
16   return(0);
17 }
```

The most interesting part of this example is the definition of the matrix `mat2` in Line 11. The even numbers from the matrix `mat` are taken as they are, whereas the odd numbers are negated. Note here, that all sub expressions in predicate position are in fact non-scalar arrays: `(mat % 2)` denotes a matrix of zeros and ones and `(mat % 2) == 0` denotes a matrix of boolean values.

Exercise 6 *What results do you expect from the following expressions:*

- `min( reshape([3,0,5], []), 42)`*?*
- `reshape([3,0,5], []) + reshape([3,0,5], [])`*?*
- `reshape([1,1], [1]) + reshape([1], [1])`*?*

Restructuring Operations

The operations to be introduced here do not compute new values at all. Instead, they are meant to create slightly differently structured arrays from existing ones. Therefore, they are applicable to arrays of all built-in element types.

take(sv, a) takes as many elements from the array `a` as indicated by the shape vector `sv`. Each element of `sv` corresponds to one axis of `a` starting from the leftmost one. For positive components of `sv`, the elements are taken from the "beginning", i.e., starting with index 0, otherwise they are taken from the "end" including the maximum legal index of the corresponding axis. All axes of `a` where there exists no corresponding element in `sv` are taken entirely.
Domain restrictions:

- `dim(sv)` $= 1$
- `shape(sv)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ...,$`shape(sv)[[0]]`$\}$: $|$`sv[[`i`]]`$| \leq$ `shape(a)[[`i`]]`

drop(sv, a) drops as many elements from the array `a` as indicated by the shape vector `sv`. Each element of `sv` corresponds to one axis of `a` starting from the leftmost one. For positive components of `sv`, the elements are dropped from the "beginning", i.e., starting with index 0, otherwise they are dropped from the "end" starting from the maximum legal index of the corresponding axis. All axes of `a` where there exists no corresponding element in `sv` are left untouched.
Domain restrictions:

- `dim(sv)` $= 1$
- `shape(sv)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ...,$`shape(sv)[[0]]`$\}$: |`sv[[`i`]]`| $\leq$ `shape(a)[[`i`]]`

tile(sv, ov, a) takes a tile of shape `sv` from `a` starting at the index specified by the offset vector `ov`. For axes where no values of `sv` or `ov` are specified these are assumed to be identical to the extent of `a` along that axis or 0, respectively.
Domain restrictions:

- `dim(sv)` $=$ `dim(ov)` $= 1$
- `shape(sv)[[0]]` $\leq$ `dim(a)`
- `shape(ov)[[0]]` $\leq$ `dim(a)`
- $\forall i \in \{0, ...,$`shape(ov)[[0]]`$\}$: `ov[[`i`]]` $\leq$ `shape(a)[[`i`]]`
- $\forall i \in \{0, ...,$`min( shape(ov)[[0]], shape(sv)[[0]])`$\}$:
  `ov[[`i`]] + sv[[`i`]]` $\leq$ `shape(a)[[`i`]]`
- $\forall i \in \{$`min( shape(ov)[[0]], shape(sv)[[0]]), ..., shape(sv)[[0]]`$\}$:
  `sv[[`i`]]` $\leq$ `shape(a)[[`i`]]`

e1 ++ e2 concatenates arrays `e1` and `e2` with respect to the leftmost axis. As in SaC all arrays are homogeneous, this requires all but the leftmost axis to be of identical extent.
Domain restrictions:

- `e1` and `e2` have to be of identical element type
- `drop( 1, shape(e1))` = `drop( 1, shape(e2))`.

rotate(ov, a) rotates the array `a` with respect to those axes specified by the offset vector `ov`. Starting from the leftmost axis, the elements of `ov` specify by how many positions the elements are rotated towards increasing indices (positive values) or towards decreasing indices (negative values). *Domain restrictions:*

- `dim(ov)` $= 1$
- `shape(ov)[[0]]` $\leq$ `dim(a)`

shift(ov, expr, a) shifts the array `a` with respect to those axes specified by the offset vector `ov`. The element positions that become "void" are filled by the (scalar) default element `expr`. Again, depending on the sign of the values of `ov` the elements are either shifted towards increasing or decreasing indices.

Domain restrictions:

- `dim(ov)` = 1
- `shape(ov)[[0]]` $\leq$ `dim(a)`
- `shape(expr)[[0]]` = `[]`

A few examples:

Listing 2.9: Restructuring Operations

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9];
7
8    mat = [ vect, vect+10, vect+20];
9    print( mat);
10
11   print( take( [2,-2], mat) );
12   print( take( [2], mat) );
13   print( take( [], mat) );
14
15   print( take( [0], mat) );
16   print( take( [2, 0], mat) );
17   print( take( [2], reshape( [3,0,5], [])) );
18
19   print( drop( [0, -1], mat) );
20
21   print( mat ++ mat);
22
23   print( rotate( [-1, 42], mat) );
24   print( rotate( [ 1], mat) );
25
26   print( shift( [0, -2], 0, mat) );
27   print( shift( [0, -22], 0, mat) );
28   print( shift( [1], 0, mat) );
29
30   return(0);
31 }
```

The applications of `take` in Lines 11-13 demonstrate, how the dimensionality of `mat` remains unaffected by the length of the first argument. Only the shape of the result and the "side" from which the elements are taken is defined by it.

The applications in Lines 15-17 demonstrate how empty arrays are dealt with in the individual argument positions. In particular from the example in Line 17 it can be seen how well the concept of having an unlimited number of different empty arrays available fits nicely into the overall framework.

The remaining examples are rather straightforward. The only aspect of interest here may be the "overflows" in the rotation and shift parameters in Lines 24 and 28, respectively.

Exercise 7 *Which of the following expressions can be reformulated in terms of* `take`*,* `++`*, and the basic operations defined in the previous parts?*

- `drop( v, a)`*?*
- `tile( v, o, a)`*?*
- `shift( [n], e, a)`*?*
- `shift( [m,n], e, a)`*?*
- `rotate( [n], a)`*?*
- `rotate( [m,n], a)`*?*

Can we define the general versions of `shift` *and* `rotate` *as well?*

Reduction Operations

The library of standard array operations that comes with the current SaC release also contains a set of functions that fold all (scalar) elements of an array into a single one. The most common ones of these are described here.

sum(a) sums up all elements of the array `a`. If `a` is an empty array, `0` is returned.
Domain restrictions: the element type has to be numerical.

prod(a) multiplies all elements of the array `a`. If `a` is an empty array, `1` is returned.
Domain restrictions: the element type has to be numerical.

all(a) yields `true`, iff all elements of `a` are `true`. If `a` is an empty array, `true` is returned.
Domain restrictions: the element type has to be boolean.

any(a) yields `true`, iff at least one element of `a` is `true`. If `a` is an empty array, `false` is returned.
Domain restrictions: the element type has to be boolean.

maxval(a) computes the maximum value of `a`. If `a` is an empty array, the minimal number of the according element type is returned.
Domain restrictions: the element type has to be numerical.

minval(a) computes the minimum value of `a`. If `a` is an empty array, the maximal number of the according element type is returned.
Domain restrictions: the element type has to be numerical.

A few examples:

Listing 2.10: Reduction Operations

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9];
7
8    mat = [ vect, vect+10, vect+20];
9
10   print( sum( mat) );
11   print( prod( vect) );
12   print( all( mat >= 1) );
13   print( any( mat > 1) );
14   print( maxval( mat) );
15   print( minval( mat) );
16
17   return(0);
18 }
```

Most of these examples, again, are fairly self explanatory. However, you may get an idea of the specificational advantages of shape-invariant programming when having a closer look at Lines 12 and 13. They demonstrate the rather intuitive style of program specifications that results from it.

Exercise 8 *All operations introduced in this part apply to* **all** *elements of the array they are applied to. Given the array operations introduced so far, can you specify row-wise or column-wise summations for matrices? Try to specify these operations for a 2 by 3 matrix first.*

2.2.2 Axis Control Notation

As can be seen from Exercise 8, without further language support, it is rather difficult to apply an array operation to certain axes of an array only. This section introduces two language constructs of SaC which, when taken together, can be used to that effect. While **Generalized Selections** are convenient for separating individual axes of an array, **Set Notations** allow to recombine such axes into a result array after applying arbitrary operations to them. However, as the two constructs in principle are orthogonal, we introduce them separately before showing how they can be combined into an instrument for **Axis Control**.

Generalized Selections

The selection operation introduced in Section 2.2.1 does not only allow scalar elements but entire subarrays of an array to be selected. However, the selection of (non-scalar) subarrays always assumes the given indices to refer to the leftmost axes, i.e., all elements with respect to the rightmost axes are actually selected. So far, a selection of arbitrary axes is not possible. As an example use-case consider the selection of rows and columns of a matrix. While the former can be done easily, the latter requires the array to be transposed first.

To avoid clumsy notations, SaC provides special syntactical support for selecting arbitrary subarrays called **Generalized Selections**. The basic idea is to

indicate the axes whose elements are to be selected entirely by using dot-symbols instead of numerical values within the index vectors of a selection operation.

Note here, that vectors containing dot-symbols are not first class citizens of the language, i.e., they can exclusively be specified within selection operations directly!

There are two kinds of dot-symbols, single-dots which refer to a single axis and triple-dots which refer to as many axes as they are left unspecified within a selection. In order to avoid ambiguities, a maximum of one triple-dot symbol per selection expression is allowed.

A few examples:

Listing 2.11: Generalized Selections

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [1,2,3,4,5,6,7,8,9];
7
8    mat = [ vect, vect+10, vect+20];
9    print( mat);
10
11   print( mat[[1]] );
12   print( mat[[1,.]] );
13   print( mat[[1,...]] );
14
15   print( mat[[.,1]] );
16   print( mat[[...,1]] );
17
18   print( mat[[1,...,1]] );
19
20   arr3d = [ mat, mat];
21   print( arr3d);
22
23   print( arr3d[[.,1]] );
24   print( arr3d[[...,1]] );
25
26   return(0);
27 }
```

The examples in Lines 11-13 demonstrate different versions for selecting the second row of the matrix `mat`. However, as the rightmost axis is to be selected, a dot-free version (cf. Line 11) suffices for this task. The selection of the second column of `mat` is shown in Lines 15 and 16.

Line 18 demonstrates that the triple-dot notation can also be successfully applied if no axis can be matched at all.

The difference between the single-dot and the triple-dot notation is shown in Lines 23 and 24. While the selection in Line 23 is identical to `arr3d[[.,1,.]]`, the one in Line 24 is identical to `arr3d[[.,.,1]]`.

Only in cases where the number of single-dots plus the number of numerical indices exceeds the number of axes available, an error message will be generated.

Exercise 9 *How can a selection of all elements of* `mat` *be specified using generalized selections? Try to find all 9 possible solutions!*

Exercise 10 *Referring to Exercise 5, can this new notation be used for selecting "over" empty axis? For example, can you specify a selection vector* <vec>*, so that* `reshape([3,0,5], [])[` <vec> `] == reshape( [3,0], [])` *holds?*

Set Notation

The means for composing arrays that have been described so far are rather restricted. Apart from element-wise definitions all other operations treat all elements uniformly. As a consequence, it is difficult to define arrays whose elements differ depending on their position within the array. The so-called **set notation** facilitates such position dependent array definitions. Essentially, it consists of a mapping from index vectors to elements, taking the general form

{ <*index_vector*> `->` <*expression*> }

where <*index_vector*> either is a variable or a vector of variables and <*expression*> is a SaC expression that refers to the index vector or its components and defines the individual array elements. The range of indices this mapping operation is applied to usually can be determined by the expression given and, thus, it is not specified explicitly.

> **A note for language design freaks:**
> *You may wonder why we restrict the expressiveness of the set notation by relying on compiler driven range detection rather than an explicit range specification. The reason for this decision is the observation that in many situations the capabilities of the set notation suffice whereas an explicit specification of ranges would obfuscate the code.*
> *Furthermore, as you will see in chapter 4,* SaC *provides a more versatile language construct for defining arrays. However, the expressiveness of that construct comes for quite some specificational overhead.*

Let us have a look at some examples:

Listing 2.12: Basic Set Notation

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [0,1,2,3,4,5,6,7,8,9];
7
8    mat = { [i] -> vect[[i]]*10+vect };
9    print( mat);
10
11   mat_inc = { iv -> mat[iv] + 1};
12   print( mat_inc);
13
14   mat_trans = { [i,j] -> mat[[j,i]] };
15   print( mat_trans);
16
17   mat_diag = { [i,j] -> where( i==j , mat[[i,j]] , 0) };
18   print( mat_diag);
19
20   return(0);
21 }
```

The set notation in Line 8 defines a vector whose components at position `[i]` are vectors that are computed from adding a multiple of `10` to the vector `vect`. The legal range of `i` is derived from the selection `vect[[i]]` yielding in fact a matrix with shape `[10,10]`. An explicit element-wise increment operation is specified in Line 11. Since the operation does not need to refer to individual axes a variable `iv` is used for the entire index vector rather than having variables for individual index components. Line 14 shows how the matrix can be transposed, and Line 17 changes all non-diagonal elements to `0`.

Exercise 11 *Which of these operations can be expressed in terms of the array operations defined so far?*

Exercise 12 *What results do you expect if* `mat` *is an empty matrix, e.g.,* `reshape( [10,0], [])`*?*

As we can see from the set notation in Line 8, non-scalar expressions within the set notation per default constitute the inner axes of the result array. This can be changed by using `.`-symbols for indicating those axes that should constitute the result axis.

A few examples:

Listing 2.13: Advanced Set Notation

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [0,1,2,3];
7
8    mat = { [.,i] -> vect[[i]]*4 + vect };
9    print( mat);
10
11   arr3d = { [i] -> vect[[i]]*16 + mat};
12   print( arr3d);
13
14   arr3d = { [.,.,i] -> vect[[i]]*16 + mat};
15   print( arr3d);
16
17   arr3d = { [.,i] -> vect[[i]]*16 + mat};
18   print( arr3d);
19
20   return(0);
21 }
```

These examples show how the result of evaluating the expression on the right of the arrow can be directed into any axes of the overall result array. As can be seen in Line 17, the axes of the expressions can even be put into non-adjacent axes of the result.

Exercise 13 *The `.`-symbol in the set notation allows us to direct a computation to any axes of the result. This is identical to first putting the result into the innermost axes and then transposing the result. Can you come up with a general scheme that translates set notations containing `.`-symbols into set notations that do without?*

Axis Control

Although generalized selections and the set notation per se can be useful, their real potential shows when they are used in combination. Together, they constitute means to control the axes a given operation is applied to. The basic idea is to use generalized selections to extract the axes of interest, apply the desired operation to the extracted subarrays and then recombine the results to the overall array.

For example, we can now easily sum up the individual rows or columns of a matrix:

Listing 2.14: Axis Control: sum

```
use StdIO: all;
use Array: all;

int main()
{
  vect = [0,1,2,3,4,5,6,7,8,9];

  mat = { [.,i] -> vect[[i]]*10+vect };
  print( mat);

  sum_rows = { [i] -> sum( mat[[i]]) };
  print( sum_rows);

  sum_cols = { [i] -> sum( mat[[.,i]]) };
  print( sum_cols);

  return(0);
}
```

Reduction operations, in general, are prone to axis control, as they often need to be applied to certain particular axes rather than entire arrays. Other popular examples are the maximum (`maxval`) and minimum (`minval`) operations:

Listing 2.15: Axis Control: max

```
use StdIO: all;
use Array: all;

int main()
{
  vect = [0,1,2,3];

  arr3d = { [i,j] -> vect[[i]]*4 + vect[[j]]*16 + vect };
  print( arr3d);

  max_inner_vects = { [i,j] -> maxval( arr3d[[i,j]]) };
  print( max_inner_vects);

  max_inner_arrays = { [i] -> maxval( arr3d[[i]]) };
  print( max_inner_arrays);

  max_outer_arrays = { [i] -> maxval( arr3d[[.,.,i]]) };
  print( max_outer_arrays);

  return(0);
}
```

In Line 8, we directly generate a 3 dimensional array from the vector `vect`. Lines 11, 14, and 17 compute maxima within different slices of that array. `max_inner_vects` is a matrix containing the maxima within the innermost vectors, i.e., the 3-dimensional array is considered a matrix of vectors whose maximum values are computed. For `max_inner_arrays`, the array is considered a vector of matrices; it contains the maximum values of these subarrays. The last example demonstrates, that outer dimensions can be considered for reduction as well.

Further demand for axis control arises in the context of array operations that are dedicated to one fixed axis (usually the outermost one) and that need to be applied to another one. Examples for this situation are the concatenation operation (`++`) and `reverse`:

Listing 2.16: Axis Control: ++, reverse

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    vect = [0,1,2,3];
7
8    arr3d = { [i,j] -> vect[[i]]*4 + vect[[j]]*16 + vect };
9    print( arr3d);
10
11   print( arr3d ++ arr3d);
12   print( { [i] -> arr3d[[i]] ++ arr3d[[i]] });
13   print( { [i,j] -> arr3d[[i,j]] ++ arr3d[[i,j]] });
14
15   print( reverse( arr3d));
16   print( { [i] -> reverse( arr3d[[i]]) });
17   print( { [i,j] -> reverse( arr3d[[i,j]]) });
18
19   return(0);
20 }
```

Line 11 shows a standard application of the concatenation of two arrays. It affects the outermost axis only, resulting in an array of shape `[ 8, 0, 0]`. The two subsequent lines show, how to apply concatenation to other axis. Essentially, the selections on the right hand sides select the sub expressions to be concatenated and the surrounding set notation glues the concatenated subarrays back together again.

The examples in Lines 15-17 show the same exercise for the operation `reverse` which reverses the order of the elements within an array with respect to the outermost axis.

Exercise 14 *The operation* `take` *is defined in a way that ensures inner axes to be taken completely in case the take vector does not provide enough entities for all axes. How can* `take` *be applied to an array so that the outermost axis remains untouched and the selections are applied to inner axes, starting at the second one? (You may assume, that the take vector has fewer elements than the array axes!) Can you specify a term that - according to a take vector of length 1 - takes from the innermost axis only?*

Exercise 15 *Can you merge two vectors of identical length element-wise? Extend your solution in a way that permits merging n-dimensional arrays on the outermost axis.*

2.2.3 Putting it all Together

The array operations presented so far constitute a substantial subset of the functionality that is provided by array programming languages such as APL. When orchestrated properly, these suffice to express rather complex array operations very concisely. In the sequel, we present two examples that make use of this combined expressive power: matrix product and relaxation.

Matrix Product

The matrix product of two matrices A and B (denoted by $A \odot B$) is defined as follows:

Provided A has as many columns as B has rows, the result of $A \odot B$ has as many rows as A and as many columns as B. Each element $(A \odot B)_{i,j}$ is defined as the scalar product of the i-th row of A and the j-th column of B, i.e., we have $(A \odot B)_{i,j} = \sum_k A_{i,k} * B_{k,j}$.

This definition can directly be translated into the following SaC code:

Listing 2.17: Matrix Product

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    id = [ [1d, 0d, 0d], [0d, 1d, 0d], [0d, 0d, 1d]];
7
8    vect = [1d, 2d, 3d, 4d];
9    mat = [ vect, vect+4d, vect+8d];
10   print( mat);
11
12   res = { [i,j] -> sum( id[[i,.]] * mat[[.,j]]) };
13   print( res);
14
15   return(0);
16 }
```

After defining two matrices `id` and `mat` in Lines 6 and 8, respectively, the matrix product `id` $\odot$ `mat` is specified in Line 12. `id[[i,.]]` selects the `i`-th row of `id` and `mat[[.,j]]` refers to the `j`-th column of `mat`. The index ranges for `i` and `j` are deduced from the accesses into `id` and `mat`, respectively. A variable k as used in the mathematical specification is not required as we can make use of the array operations `*` and `sum`.

Relaxation

Numerical approximations to the solution of partial differential equations are often made by applying so-called **relaxation methods**. These require large

arrays to be iteratively modified by so-called **stencil operations** until a certain convergence criterion is met. Fig. 2.1 illustrates such a stencil operation. A

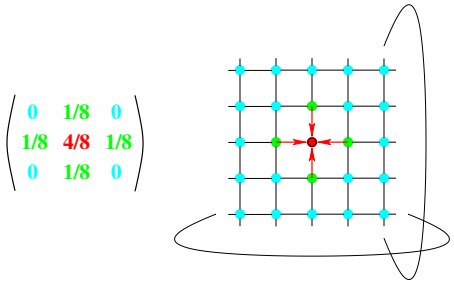

Figure 2.1: A 5-point-stencil relaxation with cyclic boundaries

stencil operation re-computes all elements of an array by computing a weighted sum of all neighbor elements. The weights that are used solely depend on the positions relative to the element to be computed rather than the position in the result array. Therefore, we can conveniently specify these weights by a single matrix of weights as shown on the left side of Fig. 2.1.

In this example, only 4 direct neighbor elements and the old value itself are taken into account for computing a new value. (Hence its name: **5-point-stencil operation**). As can be seen from the weights, a new value is computed from old ones by adding an eight-th each of the values of the upper, lower, left, and right neighbors to half of the old value.

As demonstrated on the right side of Fig. 2.1 our example assumes so-called **cyclic boundary conditions**. This means that the missing neighbor elements at the boundaries of the matrix are taken from the opposite sides as indicated by the elliptic curves.

In the sequel, we concentrate on the specification of a single relaxation step, i.e., on one re-computation of the entire array. This can be specified as a single line of SaC code:

Listing 2.18: Relaxation with Cyclic Boundaries

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    weights = [ [0d, 1d, 0d], [1d, 4d, 1d], [ 0d, 1d, 0d]] / 8d;
7
8    vect = [1d, 2d, 3d, 4d];
9    mat = [ vect, vect+4d, vect+8d, vect+12d];
10   print( mat);
11
12   mat = { [i,j] -> sum( { iv -> weights[iv] * rotate( iv-1, mat)}[[...,i,j]] ) };
13   print( mat);
14
15   return(0);
16 }
```

Line 6 defines the array of weights as given on the left side of Fig. 2.1. Our example array is initialized in Lines 8-9. The relaxation step is specified in Line 12. At its core, all elements are re-computed by operations on the entire array rather than individual elements. This is achieved by applying `rotate` for each legal index position `iv` into the array of weights `weights`. Since the expression { `iv -> weights[iv] * rotate( iv-1, mat)`} computes a 3 by 3 array of matrices (!) the reduction operation `sum` needs to be directed towards the outer two axes of that expression only. This is achieved through axis control using a selection index `[...,i,j]` within a set notation over `i` and `j`.

Exercise 16 *Another variant of relaxation problems requires the boundary elements to have a fixed value. Can you modify the above solution in a way that causes all boundary elements to be 0? [***Hint:** *You may consider the boundary elements to actually be located* **outside** *the matrix]*

Chapter 3

Basic Program Structure

This trail gives a brief introduction into the main language constructs most of which have been adopted from standard C. We assume some familiarity with standard C and, therefore, only give a quick overview and highlight the differences between SaC and C.

3.1 Lesson: Functions and their Types

3.1.1 Function Definitions

Like in other modern programming languages functions constitute the main form of structuring programs in SaC. SaC functions very much resemble their C counterparts. The most prominent difference is that SaC functions can have multiple return values, as illustrated in the following example.

Listing 3.1: Function definitions

```
1  use ScalarArith: all;
2
3  int, int divmod( int x, int y)
4  {
5    return( x/y, x%y);
6  }
7
8  int main()
9  {
10   d, m = divmod( 8, 3);
11
12   return( d);
13 }
```

A function with multiple return values, like `divmod` in the above example, has a comma-separated list of return types in front of the function name and the `return`-statement likewise contains a comma-separated list of expressions. Obviously both lists must coincide in length to make up a well-formed program. Functions with multiple return values cannot appear in expression positions in

SaC, but their results need to be directly assigned to identifiers as illustrated above.

Exercise 17 *Extend the above example program to compute the greatest common denominator of two numbers using Euclid's algorithm. In particular, use the function* `divmod`*.*

Exercise 18 *What happens, if you use the same variable name for both results of* `divmod`*?*

3.1.2 Built-in Types

SaC supports all basic types of standard C such as `int`, `float`, `double`, `char` etc.

> **A note for bit freaks:**
> *All basic types are mapped one-to-one to their C counterparts and, hence, the same rules apply to them in SaC as in C. As a consequence of this, the concrete bit widths used for representation and, hence, the range of values for integer types are platform-dependent. Although this may be considered undesireable we find it acceptable from a compatibility-with-C perspective.*

In addition to the C-inhereted types, SaC supports three more basic types: the boolean type `bool` and the integer type `byte`, both signed and unsigned. Although these are internally mapped to the same C type, on the level of SaC, there is a strict separation between them. Neither can be used in places where the other is expected. This separation also applies to the standard C types which rules out tacit coercions as one would find them in C or Java.

> **A note for language design freaks:**
> *The reader may wonder why we enforced this strict separation. The reason is two-fold: First, it prevents from accidental coercions and second, it makes type-inference more accessible for the user in the absence of explicit type declarations.*
> *Just imagine an overloading of a function* `foo` *for integer and double arguments which would yield different results depending on the type of argument. If we had implicit coercions in place it would be completely unclear how the result of* `foo( 0)` *would be computed!*

As a consequence of this strict separation, programmers need to apply some rigor when it comes to specifying constant values. They have to be attributed with the appropriate suffixes to indicate the desired runtime representation. Note here, that we adopted the suffixes and default rules from C. Here a few examples:

Listing 3.2: Element-Type segregation

```
1  use StdIO: all;
2  use Array: all;
3
4  bool foo( double x)
5  {
6    return( true);
7  }
8
9  bool bar( float x)
10 {
11   return( false);
12 }
13
14 int main()
15 {
16   /**
17    * type error: foo is not defined on int!
18    */
19   a = foo( 0);
20
21   /**
22    * correct calls:
23    */
24   a = foo( 0.0);
25   a = foo( 0d);
26
27   /**
28    * type error: bar is not defined on double!
29    */
30   a = bar( 0.0);
31
32   /**
33    * correct calls:
34    */
35   a = bar( 0f);
36   a = bar( 0.0f);
37
38   return(0);
39 }
```

Exercise 19 *How can you modify the above program in a way that allows the programmer by a simple define to switch the argument type of* `foo` *between float and double and all the calls to* `foo` *accordingly?*

*[***Hint:** *Use the C preprocessor to make the necessary modifications]*

3.1.3 Subtyping

For each basic type there is an entire hierarchy of array types that specify the shape of an array (remember that any expression in SaC denotes an array) at different levels of accurateness. Using type `int` as a running example, `int` itself denotes an integer array with rank zero, the empty vector `[]` as shape vector and a single element, in other words the equivalent of a scalar value. Then, there are (real) array types denoting arrays of rank greater than zero, e.g. `int[4]` denotes a 4-element vector while `int[10,20]` denotes a 10 by 20 element matrix.

Whereas all these types specify exact shapes, SaC also features types that solely denote the rank of some array, but leave the concrete shape open, e.g. `int[.]`

describes the type of all integer vectors (of any length) and `int[.,.]` is the type of all integer matrices. In order to support rank-invariant programming, SaC, furthermore, has types that not only abstract from concrete shapes but even from concrete ranks. These are `int[*]` which is the type of all integer arrays of any rank and shape, including rank-zero arrays (usually referred to as scalars) and `int[+]`, the type of all "true" integer arrays, i.e. arrays of rank greater than zero.

SaC defines a subtype relationship between array types in the obvious way. Figure 3.1 shows this relationship for arrays of integer elements in a graphical form. Whenever two types are in subtype relationship they are connected by a

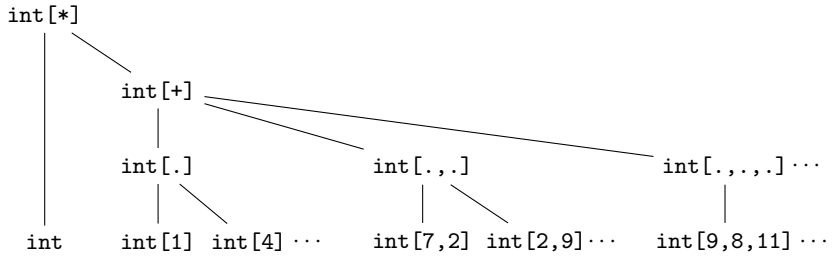

Figure 3.1: The hierarchy of array types of integer elements.

line. For example, `int[+]` is a subtype of `int[*]`, `int[.]` and `int[.,.]` are both subtypes of `int[+]` and `int[12]` and `int[42]` are subtypes of `int[.]`. As we can see, the subtyping hierarchy of SaC has exactly four levels.

3.1.4 Function Overloading

The real power of subtyping unfolds when it is combined with function overloading. It allows programmers to specify multiple functions of the same name, as long as they differ in the types or their arguments. Which definition is chosen for any given application of such a function depends on the type of the actual arguments. This combines a high degree of code reuse with the ability to later add special definitions for a few special cases.

> **A note for OO-freaks:**
> *This can be seen as a more general form of inheritance. If you restrict overloading to one argument only, say the first one, it equates to inheritance in OO languages. However, the overloading of SaC is much more powerful. Not only does it support inheritance on* **all** *arguments but it also supports overloading across different types. These features render several of the well-known OO programming pattern such as the visitor pattern superfluous in SaC. Instead, the desired overloading can be specified directly.*

Here, an example for element-type overloading:

Listing 3.3: Function overloading

```
1  use ScalarArith: all;
2  use StdIO: all;
3
4  int twice( int x)
5  {
6    return( 2 * x);
7  }
8
9  double twice( double x)
10 {
11   return( 2.0 * x);
12 }
13
14 int main()
15 {
16   a = twice( 5);
17   print(a);
18
19   b = twice( 5.9);
20   print(b);
21
22   return(0);
23 }
```

When utilising the overloading on our hierarchy of array types we can even achieve a pattern matching like programming style as demonstrated in the next example. Here, we have three instances of the function `quicksort`, one for vectors of any length, one for vectors of length one and one for empty vectors. The latter two boil down to the identity function. As a result we can safely access the first element of the argument vector `v` in the general instance because any argument vector is guaranteed to have at least two elements.

```
1  int[.] quicksort( int[.] v)
2  {
3     pivot = v[[1]];
4     ...
5  }
6
7  int[1] quicksort( int[1] v)
8  {
9    return( v);
10 }
11
12 int[0] quicksort( int[0] v)
13 {
14   return( v);
15 }
```

Exercise 20 *In a C program, functions like* `+` *can be applied to arbitrary combinations of integer and double arguments. Try to mimic that behaviour in SaC by defining a function* `cPlus` *and by overloading it appropriately.*

Exercise 21 *In OO languages inheritance cannot always be statically resolved. This leads to what is referred to as dynamic dispatch, i.e., the disambiguation of function calls at runtime. Is that required in SaC too? If so, can you come up with an example program that demonstrates this?*

3.2 Lesson: Function Bodies

3.2.1 Variable Declarations

Local variables within bodies of SaC functions are typically not declared (in contrast to C), but the SaC compiler infers proper types for local variables or yields an appropriate error message. Nevertheless, it is syntactically legal to add explicit variable declarations in SaC in exactly the same way as in C.

There is one difference to C, however: While C allows local variable declarations at the beginning of each code block and in the latest C99 standard instructions and declarations can even be interleaved, SaC only supports variable declarations on the level of function bodies, and they must precede any instruction.

Exercise 22 *Rewrite your solution to computing the greatest common denominator of two numbers from the previous exercise such that each subexpression is assigned to an identifier, i.e. flatten any nested expression. Then add explicit variable declarations for each local variable used.*

Exercise 23 *The previous exercise only used scalar types, more precisely* `int`*. What happens if you replace* `int` *in all variable declarations by its supertype* `int[*]`*?*

3.2.2 Assignments

As we have already seen in the previous trails, SaC allows for C-style assignments to variables. In contrast to C, assignments cannot be placed within expression positions and the comma-operator of C is not supported. However, the combinations of operators and assignment are the same in SaC as in C

Listing 3.4: Operator assignments in SaC

```
1  use ScalarArith: all;
2  use StdIO: all;
3
4  int main()
5  {
6    a = 42;
7    print( a);
8    a += 5;
9    print( a);
10   a -= a;
11   print( a);
12
13   return( a);
14 }
```

Note that despite the term “Single Assignment” in the name of SaC, the language actually supports repeated assignment of values to the same variable as in the above example. This seeming contradiction can be explained as follows: Each assignment opens up a new scope of an identifier bound to some value. Accordingly, the second assignmnent to `a` in the above example opens up a new

scope for a new identifier `a` that only coincidentally carries the same name as the identifier introduced in the code line before. However, because these two variables do carry the same name, the second assignment shadows the scope of the first assignment meaning that no access to the first `a` is possible any more.

3.2.3 Conditionals

In SaC, we support three forms of conditionals:

- branching with consequence only (if-then),
- branching with consequence and alternative (if-then-else) and
- conditional expressions.

All three forms use a syntax that is identical to that of C, as the following listing illustrates. Similarity with C extends to the use of curly brackets to build blocks of multiple statements and their potential absence if the condition covers a single statement only

Listing 3.5: Conditionals in SaC

```
1  use ScalarArith: all;
2  use StdIO: all;
3
4  int main( )
5  {
6    a = 5;
7    b = 7;
8    printf( "a=%d, b=%d\n", a, b);
9
10   if (a<b) a = b;
11   printf( "a=%d, b=%d\n", a, b);
12
13   if (a>=b) {
14     a = b;
15   }
16   else {
17     b = a;
18   }
19   printf( "a=%d, b=%d\n", a, b);
20
21   b = a<b ? a : b;
22   printf( "a=%d, b=%d\n", a, b);
23
24   return( b);
25 }
```

As in C all forms of conditionals can be nested in any way, and the dangling else ambiguity is resolved as in C proper.

A small difference to standard C is that the predicate expression of any conditional must be of type `bool`. There is no implicit treatment of integer values as predicates. Another subtle difference to C stems from the functional nature of SaC: a variable defined only in one branch of a conditional, will cause the SaC compiler to raise an error because the value may be undefined.

The `switch`-statement of C is currently not supported by SaC. This is not so much motivated by conceptual concerns, but rather by pragmatic considerations like the ratio between expressiveness gained and implementation effort caused.

3.2.4 Loops

SaC supports all three loop constructs of standard C: `while`, `do` and `for` with the familiar syntax, as illustrated by the following code fragment.

Listing 3.6: Loops in SaC

```
1  use ScalarArith: all;
2  use StdIO: all;
3
4  int main()
5  {
6    a = 10;
7
8    while (a>0) {
9      a = a - 2;
10     print( a);
11   }
12
13   do {
14     print( a);
15     a = a + 1;
16   }
17   while (a<7);
18
19   for (i=1,j=2 ; i+j < 42 ; i++, j++) {
20     a *= 2;
21     print( a);
22   }
23
24   return( a);
25 }
```

In analogy to conditionals, the loop predicate expression must be of type `bool`. Note that SaC does even support the comma operator in `for`-loops (though not in general terms as pointed out before).

These C-style loop constructs can make code look very imperative. Despite these syntactic similarities, always bear in mind that SaC loops are (only) syntactic sugar for equivalent tail-end recursive functions. While the functional semantics almost completely coincides with the pragmatic expectations of a C programmer, some subtle issues may arrise concerning the definedness of variables. For example, the SaC compiler would complain about the above example saying that the variable `b` in the `while`-loop may be used uninitialised if it is not defined before. This is because the compiler assumes that the body of a `while`-loop may not be executed at all. Of course, you may know better, but the SaC compiler at the moment makes no particular effort to prove this fact when it analyses the definedness of variables.

3.2.5 Explicit Control Flow Manipulation

The control flow manipulation statements of C, i.e. `goto`, `break` and `continue`, as well as labels are not supported by SaC. This is due to the fact that SaC is indeed a functional language and as such there is actually no control flow, even though the C-like syntax suggests one.

3.3 Lesson: Advanced Topics

3.3.1 User-defined Types

SaC allows programmers to define their own types using a syntax that is identical to C.

Listing 3.7: User-defined types

```
1  use StdIO: all;
2
3  typedef int myint;
4  typedef float[100,100] real_matrix;
5  typedef double[2] complex;
6
7  int main()
8  {
9    complex   c;
10   double[2] d;
11
12   c = (:complex) [1.2,2.3];
13   d = (:double[2]) c;
14   print( d);
15
16   return( 0);
17 }
```

Following the keyword `typedef` we have the defining type followed by the defined type name. Note that in contrast to C, defining type and defined type are not considered synonyms. Types like `double[2]` and `complex` are distinguished properly and a function that expects a value of type `double[2]` as an argument will not accept a value of type `complex` instead.

SaC requires explicit type casts to change the type of a value from the defining type to the defined type or vice versa as in the following example.

Note that for the time being any defining type in a type definition must exactly specify some shape; the various less specific types are not supported. While this looks incomplete at first glance, it is noteworthy that such type definitions would immediately lead to arrays of non-uniform shape, e.g. a matrix whose rows have different length. There is no doubt that this would be an extremely powerful extension to the homogeneously shaped arrays that SaC supports today, but it would likewise require a non-trivial extension of the code generator and runtime system that we leave for future research.

3.3.2 Type Conversions

SaC uses C-like cast expressions to change the type of an expression whenever the data repesentation remains unaffected, i.e. between user-defined types and their defining types or vice versa. In contrast to C, SaC does not use cast expressions to actually change data representations, e.g. when converting from an integer type to a floating point type, between floating point types of different precision or between integer types of different bit width. For all these purposes SaC uses dedicated conversion functions to express the fact that such conversions actually require an operation performed at runtime rather than just changing the type interpretation of a value.

These conversion functions are named `tobool` and `tochar` for converting into non-numerical values. For all numerical types these functions are named "to" plus an optional "u" for unsigned integer types followed by the first letter of the type name ("ll" in the case of `long long int`). The following example illustrates type conversions in SaC.

Listing 3.8: Type conversions

```
1  use ScalarArith: all;
2  use StdIO: all;
3
4  int main()
5  {
6    double x;
7    float y;
8    int z;
9
10   x = 2.3;
11   print(x);
12
13   y = tof( x);
14   print(y);
15
16   z = toi( y);
17   print(z);
18
19   return( z);
20 }
```

Chapter 4

With-Loops

This trail aims at providing a hands-on introduction to the key language construct of SaC: the With-Loop. It constitutes the generalisation of the set-expression as introduced in the lesson 2.2 on **Shape-Invariant Programming** and can be seen as a shape-invariant form of the map-reduce template or the array comprehensions found in other functional languages. However, in contrast to these, the With-Loop was carefully designed to enable radical code optimisations as well as compilation into high-performance, concurrently executable code.

> **A note for language design freaks:**
> *In fact, almost all array operations introduced in earlier trails are defined by With-Loops within the standard library. This design combines two maior advantages:*
>
> - **better performance**, *as the conformity enables optimisations to be more generally applicable, and*
> - **increased flexibility**, *as the user can modify the definition of all standard operations.*

The introduction of With-Loops comes in a single lesson which step-wise introduces all features and variants of With-Loops in SaC.

4.1 Lesson: With-Loop Basics

4.1.1 Basic Components

Generally, With-Loops are composed of three different components:

- sets of index vectors (referred to as **generator-ranges**),
- functions that map index vectors to arbitrary values (**generator-expressions**), and

- combining operations (**with-loop operators**) that take such values and construct arrays from them.

In its simplest form, a With-Loop contains one component of each kind. It then maps the function defined by the generator-expression to all index vectors from the generator-range in a data-parallel fashion. This leads to a set of index-value-pairs which are combined into a result array by means of the given with-loop operator. Let's have a look at a simple example:

Listing 4.1: Simple With-Loop

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = with {
7          ( [0] <= iv < [5]) : 42;
8        } : genarray( [7], 0);
9
10   print(a);
11
12   return(0);
13 }
```

Here, the With-Loop in lines 6-8 computes the vector `[ 42, 42, 42, 42, 42, 0, 0]`. The generator-range is specified by the code snippet in round brackets in line 7: `( [0] <= iv < [5])`. It denotes the set of vectors { `[0]`, `[1]`, ..., `[4]` }. The generator-expression here is the constant `42`. Hence, the mapping function f_{map} maps any index vector `iv` into `42`, i.e., we have f_{map}(`iv`) =`42`. Finally, the with-loop operation is specified as **genarray([7], 0)**. This operation computes an array of shape `[7]` where

$$\texttt{a[iv]} = \begin{cases} f_{map}(\texttt{iv}) & \text{iff } \texttt{iv} \in \{\texttt{[0]},\texttt{[1]},...,\texttt{[4]}\} \\ \texttt{0} & \text{otherwise.} \end{cases}$$

Generator-ranges and **generator-expressions** always occur in pairs. Jointly they form a syntactical unit, which we refer-to as **generator**. As we will see later, With-Loops can contain arbitrary numbers of generators. They are enclosed in curly brackets.

Exercise 24 *What result do you expect if we eliminate the generator from the above example?*

What results do you expect if we modify the generator-range in the above example into:

- `( [-2] <= iv < [3])`*?*
- `( [0] <= iv < [8])`*?*
- `( [6] <= iv < [5])`*?*
- `( [8] <= iv < [5])`*?*
- `( [6] <= iv < [0])`*?*

*[***Hint:** *You should compile these examples with the option* `-check c` *being enabled]*

4.1.2 Generator Ranges

SaC offers quite some flexibility when it comes to specifying generator ranges. First of all, the use of index vectors in the bounds enables the convenient specification of n-dimensional index ranges. Let us look at a few examples for the 2-dimensional case:

Listing 4.2: Generator Range Specifications

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = with {
7          ( [0,2] <= [i,j] < [5,6]) : 42;
8        } : genarray( [5,6], 0);
9    print(a);
10
11   shp = [5,6];
12   a = with {
13         ( 0*shp <= iv < shp) : 42;
14       } : genarray( shp, 0);
15   print(a);
16
17   a = with {
18         ( . < iv < .) : 42;
19       } : genarray( [5,6], 0);
20   print(a);
21
22   a = with {
23         ( . <= jv=[x,y] <= [4,5]) : 42;
24       } : genarray( [5,6], 0);
25   print(a);
26
27   a = with {
28         ( . <= [i,j] <= . step [1,4] ) : 42;
29       } : genarray( [5,6], 0);
30   print(a);
31
32   a = with {
33         ( . <= [i,j] <= . step [2,4] width [1,2] ) : 42;
34       } : genarray( [5,6], 0);
35   print(a);
36
37   return(0);
38 }
```

As we can see from the first With-Loop in lines 6-8, a vector of scalar indices can be used where we previously used the variable `iv` to denote the entire index vector. In cases where the dimensionality of the With-Loop is statically fixed, this sometimes comes in handy. However, if we want to adopt a more generic, shape-invariant programming style, it becomes mandatory to use vectors for the index variable as well as for the bounds.

Note here, that the ability to use vectors rather than componentised indices and bounds is absolutely crucial here! It constitutes **the** *enabling factor for specifying With-Loops in a shape-invariant style as the length of those vectors may remain unknown until runtime. It is this particular feature that sets the With-Loops apart from most conventional language constructs for data-parallel array operations.*

The With-Loop in lines 12-14 demonstrates a typical case where the dimensionality of the resulting array is solely determined by a vector (here `shp`). A slightly more elegant way for the most frequent case is the use of a syntactical shortcut supported by SaC. The symbol "`.`" can be used in the position for the lower and upper bound, denoting the lowest legal index and the highest legal index into the array to be created, respectively. This is examplified in the With-Loop in lines 17-19. Note here, that this generator-range does **not** cover the entire legal index space of the resulting array! As the "`.`" allways represents legal indices, we have to make sure that we use `<=` on both sides if we want to cover the entire range. The example presented here, excludes the extreme cases and, thus, covers all inner elements of the resulting array only.

Exercise 25 *What happens if the length of the vectors within the generator-range or the shape expresssion in the* `genarray`*-operation do not match? [***Hint:** *You should compile these example with the option* `-check c` *being enabled]*

The With-Loop in lines 22-24 demonstrates how both, a vector version of the index vector and scalarised versions can be made available for the generator-expression. It also examplifies that a mix of the `.`-symbol and explicit expressions can be used for the bounds.

The remaining two With-Loops demonstrate the ability to specify rectangular grids of indices. The vector that follows the keyword `step` specifies the stride of the reoccurence pattern per axis. As a consequence, the With-Loop in lines 27-29 computes an array whose every fourth column is `42` starting with the very first one.

The use of the vector after the keyword `width` enables the programmer to denote more than one index per stepping period. The With-Loop in lines 32-34 computes a matrix where 1x2 blocks of the value `42` are placed in the upper left corner of each 2x4 grid of the resulting array.

Exercise 26 *Can you achieve the same result array as the last With-Loop of the above examples* **without** *using the step/width facility? [***Hint:** *The solution may be surprisingly short!]*

4.1.3 Generator Expressions

As we have seen in the previous sections, each generator expression implicitly defines a mapping function from indices to expressions. The parameters of these functions are derived from the index variables introduced in the associated generator range specifications. More complex generator expressions can be specified

by assignment blocks that can be inserted between a generator range and the associated generator expression. Here a few examples:

Listing 4.3: Non-trivial generator expressions

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = with {
7          ( [0,1] <= [i,j] < [6,6]) {
8            div, mod = divmod( i, j);
9          } : mod;
10       } : genarray( [6,6], 0);
11   print(a);
12
13   a = with {
14         ( . <= [i,j] <= . ) : ( i==j ? 1: 0) ;
15       } : genarray( [6,6], 0);
16   print(a);
17
18   a = with {
19        ( . <= [i] <= .) {
20           mval = i;
21           val = sum( with {
22                        (. <= [i] <= .) : i
23                      } : genarray( [mval], 0));
24        } : val;
25       } : genarray( [6], 0);
26   print(a);
27
28   return(0);
29 }
```

The first With-Loop in lines 6-10 shows a typical scenario. The function divmod reurns two values rather than just one. Rather than defining an explicit mapping function that passes on the desired return value, we can specify this selection directly.

The With-Loop in lines 13-15 demonstrates how non-trivial expressions can be used even without necessitating the introduction of an assignment block.

The scope of variables that are defined in such an assignment block is strictly local to that block. Such a varable can neither be referenced within other generators of the same With-Loop nor outside of the With-Loop. Note, however, that With-loops can be arbitrarily nested. An example for such a nesting is schon in the With-Loop in lines 18-22 of the examples above.

Exercise 27 *What do you expect to happen, if a variable that is defined in such an assignment block has the same name as the index variable? Where is the "modified" version observable?*

Why can the variable `mval` *in the example above be safely replaced by* `i`*?*

4.1.4 Reductions and further With-Loop Operations

Besides the genarray with-loop operator described so far, SaC supports a few more. These are:

- a modarray operator which "modifies" an existing array, and
- two fold operators that enable the specification of reduction operations

The modarray variant is very similar to the genarray variant. The only difference is that neither the shape of the result nor a default element for unspecified index positions are explicit. Both of these are taken from a specified array that serves as a template. The second With-Loop in lines 11-13 of the example below demonstrates this. Here, a new array `b` is computed from the array `a` by negating each second element of `a`. As in lesson 2.1 on arrays as data, printing `a` in line 15 shows that in fact two different arrays have been created.

Listing 4.4: Further With-Loop operators

```
1  use StdIO: all;
2  use Array: all;
3
4  int main()
5  {
6    a = with {
7          ( . <= [i] <= .) : i;
8        } : genarray( [6], 0);
9    print(a);
10
11   b = with {
12         ( . <= iv <= . step [2] ) : -a[iv];
13       } : modarray(a);
14   print(b);
15   print(a);
16
17   c = with {
18        ( [0] <= iv <= [5]) : a[iv];
19       } : fold( *, 1);
20   print(c);
21
22   d = with {
23        ( [0] <= iv <= [5]) : a[iv];
24       } : foldfix( *, 1, 0);
25   print(d);
26
27   return(0);
28 }
```

The two final examples demonstrate how reduction operations can be specified in SaC. The first variant (lines 17-19) is the standard variant for reductions. It requires the specification of a folding function (`*` in this case) and of a neutral element (`1` here). As with the earlier variants, all index vectors from the generator range are mapped according to the generator expression. Subsequently all computed values are folded using the specified folding function. Note here, that no particular folding order is guaranteed! In order to obtain deterministic results on a multicore machine this requires the specified folding operation to be both, associative and commutative.

The final With-Loop in lines 22-24 is a slight variant of the fold version. It stems from the observation that some reductions can be shortcut when a certain fixpoint value has been met. In the given example of multiplication this is the value `0`. Whether the underlying implementation makes use of this extra

information or not is not specified. The computational result of both, the fold and the foldfix variant are the same, provided that the specified fixpoint value infact constitutes one for the specified folding operation.

Chapter 5

Working with Modules

This chapter explains the basic principles of using modules in SaC. Prior to modules, a short introduction into SaC name spaces is given.

5.1 Name Spaces

In general, name spaces are used to extend the set of possible identifiers and thus inhibit potential name clashes. SaC supports multiple name spaces, although these are not explicitly defined by the programmer. Instead, every module and program has its own name space. The name space of a module is denoted by its name, a program uses `main` as its name space identifier. As an example of using name spaces, look at the modified *hello world* listing below.

Listing 5.1: Hello world program with name spaces

```
1 int main()
2 {
3   StdIO::printf( "Hello World!\n");
4   return(0);
5 }
```

Instead of an import statement, a qualified function name is used. A qualified identifier always consists of a name space identifier, followed by a double colon and the unqualified identifier, in this case a function identifier. Besides of functions, fully qualified names may as well be used for types and global objects.

As it would be bothering to precede each identifier with the name space it belongs to, SaC supports multiple ways to automatically decide the right name space and generate a fully qualified identifier implicitly (or internally).

5.2 Use Statements

To simplify the use of identifiers from other modules, SaC allows us to specify a search space whose identifiers can be used in an unqualified fashion. By default,

this search space contains all identifiers from the current name space. To add a complete name space to the current search space, you may use the statement `use` *name space* `all;` where *name space* gives the name of a name space. Using this technique, the hello world example can as well be expressed as follows:

Listing 5.2: hello world with use statement

```
1 use StdIO: all;
2
3 int main()
4 {
5   printf( "Hello World!\n");
6   return( 0);
7 }
```

However, an identifier is not allowed to have multiple occurrences within the search space as far as types and global objects are concerned. For function symbols the same holds modulo overloading based on parameter types.

To further avoid name clashes, SaC supports a more specific way to define search spaces. Instead of the keyword `all`, a comma separated list of identifiers can be given. The following version of hello world uses a more specific use statement:

Listing 5.3: hello world with specific use statement

```
1 use StdIO: {printf};
2
3 int main()
4 {
5   printf( "Hello World!\n");
6   return( 0);
7 }
```

In this example, only `printf` is made available to the search space and can thus be used without explicitly specifying its name space. In some occasions it can be useful to add all identifiers except a given set to the search space. Consider a module `FastIO` reimplementing all functions of `StdIO`. To use `FastIO` except the `printf` function, but `StdIO::printf` one might write:

Listing 5.4: hello world with use all but statement

```
1 use FastIO: all except {printf};
2 use StdIO: {printf};
3
4 int main()
5 {
6   printf( "Hello World!\n");
7   return( 0);
8 }
```

This adds all identifiers from `FastIO` to the current search space except `FastIO::printf`. This allows the function `printf` to be imported from module `StdIO`.

5.3 Import statement

So far, we added identifiers to the search space of the SaC compiler to avoid explicitly specifying their name spaces each time they are referenced. As function signatures have to differ in number of arguments or their types, the `use` statement prevents overloading of functions by shape across module boundaries. In fact, SaC only supports overloading by shape within a single name space. Otherwise, the meaning of a fully qualified identifier could differ when being used in different scopes. To nonetheless allow for successive overloading in separate modules, SaC provides a mechanism for cloning functions using the `import` statement.

Listing 5.5: hello world with import statement

```
1  import StdIO: {printf};
2
3  int main()
4  {
5    main::printf( "Hello World!\n");
6    return( 0);
7  }
```

In this example, the statement`import StdIO: printf` creates a (conceptual) copy of the function `printf` in the current name space `main`. Consider a module `foo` containing a function `int[*] bar( int[*])`. This function can now be overloaded as follows:

Listing 5.6: bar overloading

```
1  import foo: {bar};
2
3  int[+] bar(int[+] x)
4  {
5    ...
6  }
7
8  int main()
9  {
10   ...
11
12   y = bar(x);
13
14   ...
15 }
```

Within the name space `main` there are two instances of `bar`, the one imported from `foo` and the one defined within `main` itself. Keep in mind, that there is a conceptual copy in `main`, so both are defined in `main` and so overloading can take place.

Be careful when importing types, as an imported type is regarded as different to its origin type by the type system as they were defined in different name spaces. However, you might still exchanges values between both types by means of a cast expression.

5.4 Putting It Together

Both types of module statements can be mixed in any possible way, as long as no name clash is introduced by them. Always keep in mind that an `import` statement adds all identifiers to the current name space and thus to the current search space, as well. The following example creates a name clash by using and importing the same identifier:

Listing 5.7: name clash example

```
1 use StdIO: all;
2 import StdIO: {printf};
3
4 int main()
5 {
6   printf("Never see this!\n");
7
8   return( 0);
9 }
```

Here, `printf` is imported to the current name space, so there is an identifier `main::printf`, which is part of the compiler search space. Furthermore, the `use` statement adds the complete name space of module `StdIO` to the current name space, especially the identifier `StdIO::printf`. Thus there are two identifiers with the same unqualified name within the search space. To solve this, a restriction to the `use` statement can be used.

Listing 5.8: name clash example

```
1 use StdIO: all except {printf};
2 import StdIO: {printf};
3
4 int main()
5 {
6   printf("Hey, it works!\n");
7
8   return( 0);
9 }
```

In this example, the identifier `StdIO::printf` is no longer added to the compiler search space and thus no name clash originates.

5.5 Implementing Modules

A SaC module implementation essentially looks just like a program, being a collection of type, global object and function definitions. Unlike a program, a module starts with the key word `module` followed by the module name, i.e. the name space, and a semicolon.

Listing 5.9: module implementation example

```
1 module mymod;
2
3 provide all except {foo};
```

```
4  export {foo};
5
6  int foo( int x)
7  {
8    return( ...);
9  }
10
11 int bar( int x)
12 {
13   return( ...);
14 }
```

The more interesting aspect of a module (name space) is the question which symbols (types, global objects and functions) are made available outside the module and which are merely accessible internally within the module itself. Two kinds of statements using the key words `provide` and `export` provide fine-grained control over the availability of symbols outside the current module. By default any symbol defined in a module is only accessible in the module itself. Using the `provide` statement all or selected symbols can be made available to be "used" by other modules or programs. With the `export` statement symbols are made available for either "use" or "import" by other modules or programs. The syntax of `provide` and `export` statements is very similar to that of the corresponding `use` and `import` statements. Either all symbols are provided/-exported uniformly, or a specific list of symbols is concerned, or all but a given list of symbols.

Chapter 6

Case Studies

This trail will illustrate how to use SaC for real-world applications. The following exercises make use of the techniques introduced in this tutorial so far. Working through this trail shall give you some more hands-on experience with the language itself and at the same time show you how to employ SaC to solve your every-day problems in an efficient way.

6.1 Lesson: Image Processing

Digital image processing spans across a vast area of applications. Ranging from digital photography over astronomy to surgery-assisting medical imaging, most applications still share as their underlying theory some sort of basic signal processing on two-dimensional (and potentially higher-dimensional) signals. It is common to work with discretised signals, i.e., an approximation of the originally analogue, continuous version of the signal.

In this exercise we will focus on two basic image filters on static, two-dimensional, 8-bit gray-scale images with a resolution of $x_{max} \times y_{max}$, such that an image is a 2D function

$$\begin{aligned} S : \mathbb{N} \times \mathbb{N} &\rightarrow \mathbb{N} \\ S(x,y) &= \begin{cases} v_{xy}, & \text{if } x < x_{max} \wedge y < y_{max} \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

where $\forall x, y : 0 \leq v_{xy} \leq 255$. Informally speaking, the parameters x and y determine the position within the image and v_{xy} determines the gray-scale value of the *pixel* at this position.

A common technique to modify an image is to apply a *filter* of desired properties to it. In many cases, an image filter F again is a two dimensional signal as the one described above. The application of a filter to an image is

expressed by the 2D (discrete) *convolution* $F \otimes S$ of these two signals:

$$F(x,y) \otimes S(x,y) \quad = \quad \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} F(j,k)S(x-j,y-k)$$

With this simple form of filtering, a wealth of image modifications are expressible. Depending on the choice of the filter mask we can achieve effects such as smoothing, sharpening, edge detection, embossing and many more.

In SaC we can represent these signals as 2D arrays where x and y are the column and row indices and each element corresponds to one pixel. The application of a filter to an image may then be expressed using a stencil operation as it was introduced in an earlier part of this tutorial (Section 2.2): The filter mask is positioned with its center point over each pixel of the original image. At each position, the pixel value of the image and its corresponding value of the filter mask are multiplied. The pixel value at the same position in the result image is determined by the sum of these products (weighed sum).

Exercise 28 *The Sobel operator is an edge-detection filter that computes the gradient image from an input signal. The filtering process consists of three steps: In the first step, horizontal edges are detected, in the second step vertical edges are detected, and in the final step, both sub-results are combined to the resulting gradient image. The first two steps are in fact two independent operations, each of which requires its own individual filter mask:*

$$F_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \qquad F_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Applying these two masks to an input image S yields two sub-results, one highlighting horizontal edges ($S_x = F_x \otimes S$), the other highlighting vertical edges ($S_y = F_y \otimes S$). We compute the final gradient image by adding the two sub-results $S_{xy} = S_x + S_y$. As S_{xy} may contain elements with a value greater than 255, we need to cap each pixel value at the maximum allowed value of 255. See Fig. 6.1 for an example of how an image and its gradient image may look like.

Quite the opposite to edge detection, a smoothing or blurring filter is used to make edges appear less promiment. Commonly applied filters of such kind are the 3×3 and 25×25 Gaussian blurring stencils:

$$G_9 = \frac{1}{15} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 2 & 3 & 2 \\ 1 & 2 & 1 \end{pmatrix} \qquad G_{25} = \frac{1}{331} \begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 20 & 33 & 20 & 4 \\ 7 & 33 & 55 & 33 & 7 \\ 4 & 20 & 33 & 20 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix}$$

Write a program that applies these operators to a given image and outputs the result for later use:

- *Write a function* **apply** *that takes an image and a filter mask as input and returns the convolution of the two. Use either with-loops or axis-control notation to achieve this. Take care of boundary conditions when implementing the convolution.*
- *Write a function* **sobel** *that takes an image as input and applies F_x and F_y to the image. Furthermore, add S_x and S_y with capping so that this function returns the final result of the Sobel operation.*
- *Write functions* **gauss9** *and* **gauss25** *that take an image as input and apply G_9 and G_{25} to the image.*
- *Write a* **main** *function that reads in an image from* **stdin**, *calls* **sobel**, **gaussBlur9** *and* **gaussBlur25** *on this image and then write the results back to* **stdout**.

Yoy may want to use the following skeleton for your program:

Listing 6.1: Skeleton of sobel.sac

```
1  use Structures: all;
2  use StdIO: all;
3  use Clock: {time, gettime, difftime};
4
5  /*
6   * Helper to write matrix to stdout in plain matrix format
7   * The produced output (let's say m.dat) can be used with gnuplot like so:
8   *
9   * gnuplot> plot "m.dat" matrix with image
10  */
11 void writeMatrix( int[.,.] m)
12 {
13   for( y=0; y<shape(m)[1]; y++) {
14     for( x=0; x<shape(m)[0]; x++) {
15       printf( "%d ", m[[x,y]]);
16     }
17     printf( "\n");
18   }
19 }
20
21 /*
22  * Convolution with given mask
23  */
24 inline
25 int[.,.] apply( int[.,.] mask, int[.,.] img)
26 {
27   /* implement convolution here */
28 }
29
30 int[.,.] sobel( int[.,.] img)
31 {
32   SY = [
33         [1, 2, 1], [0, 0, 0], [-1, -2, -1]
34        ];
35   /* complete this function here */
36 }
37
38 int[.,.] gaussBlur9( int[.,.] img)
39 {
40   /* complete this function here */
```

```
41 }
42
43 int[.,.] gaussBlur25( int[.,.] img)
44 {
45   /* complete this function here */
46 }
47
48 int[*], time timestamp( int[*] img)
49 {
50   return( img, gettime());
51 }
52
53 int main()
54 {
55   fprintf( stderr, "\nReading image in Fibre format...\n");
56   img = FibreScanIntArray( stdin);
57
58   fprintf( stderr, "\nApplying filter ...\n");
59   img, start = timestamp( img);
60   img = sobel( img);
61   img, end = timestamp( img);
62
63   /* Call the other filters here */
64
65   fprintf( stderr, "\nFinished filtering after %fs\n", difftime( end, start));
66   fprintf( stderr, "\nWriting result image ...\n");
67   writeMatrix( img);
68
69   return( 0);
70 }
```

Compile your source code with and without the ***-mt*** *option for multi-threaded execution and experiment with various thread counts. On faster machines it might be necessary to apply the filter multiple times (copy and paste line 50 a couple of times) to see measurable speed-ups.*

NB: The Fibre format encodes, in addition to the raw data, shape information. By using FibreScanIntArray this program is able process 2D images of any size and it is not fixed to statically known input sizes.

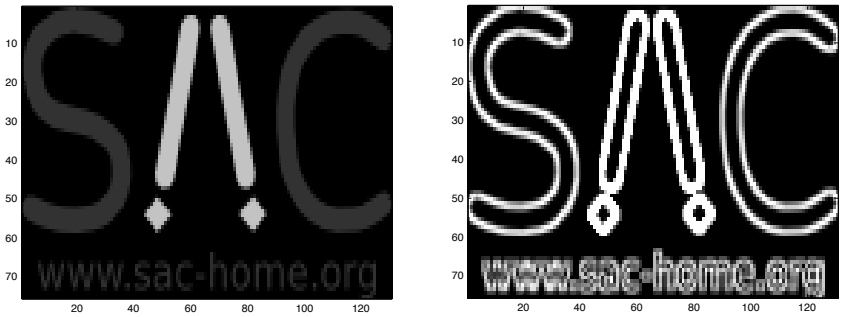

Figure 6.1: The SaC logo before (left) and after (right) edge detection

6.2 Lesson: Computing Mandelbrot Images

Exercise 29 *This exercise is about creating a basic implementation for computing mandelbrot images.*

1. *To get started, you may want to use the files* `mandelbrot_start.sac` *and* `Fractal_tier1.sac` *which, in essence, contain the IO-code for visualising the mandelbrot pictures. A first running version can be obtained by implementing the missing function bodies in* `Fractal_tier1_empty.sac`*:*

   - `escapeTime` *which implements the iteration on arrays of complex numbers, and*
   - `genComplexArray` *which computes a two-dimensional array of complex numbers that represent a descritisation of* $\mathbb{C}^2$*.*

2. *Waiting for the final picture can be rather unpleasant if it is not clear whether the chosen fraction of* $\mathbb{C}^2$ *yields an interesting picture and the iteration limit is high. therefore, as a first extension, try to modify the* `main` *function in* `mandelbrot_start.sac` *so that it computes the mandelbrot picture with increasing resolution without changing the overall size of the picture.*

   Compute resolutions `[5,5]`*,* `[10,10]`*, ...,* `[320,320]` *and display them consecutively in a* `[320,320]` *display by replicating the found values accordingly.*

   Hint: define a function `stretchRgb` *which takes an array of type* `color[.,.]` *and an integer stretching factor* `stretch` *and replicates each element of the array into a square of shape* `[stretch, stretch]`*.*

3. *The function* `intArrayToMonochrome` *maps all escape values into a color by means of a clut. Can you express this operation without a With-Loop?*

   Hint: You may find inspiration in one of the earlier tasks!

4. *Try using the compiler option* `-mt` *to experiment with multi-core machines!*

5. *Try using the compiler option* `-cuda` *to experiment with graphics cards!*

6. *Try using the compiler option* `-b11:cyc` *to inspect the high-level, optimised intermediate code.*

Exercise 30 *In this exercise, we improve the way the colours are chosen in order to obtain smoother transitions. We will use a common approach referred to as* normalized iteration counts*. A normalized iteration count for a point in the complex plane* z *is computed by using the iteration count* t *and the value at that position* z_t *during the final iteration. Using these two values, the normalized iteration count is defined as* $t_n := (t+1) - log_2(log_2(|z_t|))$ *for those values that escape and as* 0 *otherwise.*

1. *The module* `Fractal_tier2.sac` *contains stubs for the missing functionality required in this exercise:*

   - `normalizedIterationCount` *which implements the normalisation of iteration counts by taking the final computed value into account.*

     Hint: The function `escapeTime` *only computes the number of iterations before the value at a given position escapes. To normalize these, the final value at that position is required, as well. For this, we have provided a function* `escapeValue`*.*

   - `doubleArrayToRGB` *maps the normalized iteration counts, which are double values, to an RGB colour-triple. To derive an RGB value, first scale all values such that they are in the interval [0:360). This value can then be used as the hue in the HSB model.*

     Hint: The module `Color8` *defines a function* `Hsb2Rgb` *that converts a HSB color description into its corresponding RGB representation.*

Exercise 31 *In this exercise, we apply the filters from the previous case study to the mandelbrot pictures. As before, we have provided stubs for the missing functionality. For this exercise these can be found in the file* `Stencil_tier3.sac`*.*

1. *Implement the functions* `apply`*,* `sobelEdges`*,* `gaussBlur` *and* `gaussBlur25` *as described in the previous case study.*

2. *The three filters described above only operate on a single channel, e.g., a gray-scale image. To lift these to colour images, implement a corresponding function for each filter that applies the filter to each colour separately.*