Computer Science — University of Amsterdam

Master Computer Science

University of Amsterdam

Nested Arrays in Single Assignment C

Roeland Jago Douma
0518476
r.j.douma@uva.nl

October 11, 2011

Supervisor: dr. Clemens Grelck

Nested Arrays in Single Assignment C

Master's Thesis

written by

Roeland Jago Douma

under the supervision of **dr. Clemens Grelck**,
and submitted in partial fulfilment of the requirements for the degree of

M.Sc. in Grid Computing

at the *University of Amsterdam*

Date of public defense:
October 19, 2011

Members of the Thesis Committee:
dr. Clemens Grelck
dr. Inge Bethke
dr. Alban Ponse

UNIVERSITY OF AMSTERDAM

Abstract

In many languages when one talks about arrays this means rectangular arrays. However, there are many problems that are not rectangular and it is often nonlogical to describe them using rectangular arrays. We will call these non rectangular arrays: irregular arrays.

In this thesis we introduce an implementation for irregular arrays, called nested arrays, in Single Assignment C (or SAC for short). We discuss the design space of irregular arrays and extend the SAC language, and the SAC compiler to support irregular arrays.

As a result we show that for SAC programs that have irregular data the use of nested arrays can lead to a significant reduction in memory requirements, while at the same time providing substantial speedup.

Acknowledgements

I would firstly like to thank my supervisor Clemens Grelck for his invaluable guidance and insights throughout this project, without which this thesis could not have been done.

Furthermore I would also like to thank all the SaC developers for answering my questions and actively helping me find solutions to the problems I encountered with the SaC compiler.

Moreover, I am also grateful to the Computer Systems Architecture (CSA) research group for providing me with a workplace and, more importantly, for the great atmosphere to do my work in.

Finally I would like to thank Maureen Arntz for her support during my late night coding session.

Contents

CHAPTER 1

Introduction

Single Assignment C [13] (SaC for short) is a purely functional programming language [15] with an ANSI-C [19] like syntax. SaC is designed for computationally intensive applications which can be found in many different fields such as scientific computing, image processing, simulation etc. One of the main design goals of SaC is to provide fast run times while at the same time offering a coding style with a high level of abstraction. Since SaC is a purely functional programming language there is no notion of a control flow. Instead SaC is based on the principle of context free substitution. For a more in depth introduction into SaC see Section 2.

SaC is, like APL [18, 10] and MatLab [20], an array language. This means that in SaC arrays are considered first class objects. Or even more abstract all the values in SaC are considered arrays. Even scalar values are considered arrays. Since all values are arrays this means that they can be passed as arguments to functions and returned from functions.

Each array has a rank and a shape. The rank of an array is the number of dimensions. For example a vector has a rank of one and a matrix has a rank of two.

The shape of an array is the number of elements in each dimension. The shape of an array is in general represented as a vector, called the *shape vector*. The i^{th} element of the *shape vector* is the number of elements in the i^{th} dimension of the array.

As an example consider the vector

$$\begin{pmatrix}1 & 2 & 3 & 4 & 5\end{pmatrix}.$$

The rank of this vector is one and the shape is represented by the vector $\begin{pmatrix}5\end{pmatrix}$. The number of elements in an array can be calculated by multiplying the number of elements in each dimension. Take the matrix

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

it has a rank of two and the vector describing the shape is $\begin{pmatrix}2 & 4\end{pmatrix}$. This means that the matrix has 2 rows and 4 columns. The number of elements in this matrix is $2 \times 4 = 8$.

In this thesis when talking about vectors this means row vectors. As a result an n by m matrix has n rows and m columns. And an n by m by p tensor has n planes, m rows and p columns, etc.

We will use the term multidimensional array to denote dense rectangular arrays. Note that any scalar or vector that only contains scalar elements is then a multidimensional array by definition.

Consider the following vector of vectors:

$$\begin{pmatrix}\begin{pmatrix}1 & 2 & 3 & 4\end{pmatrix} & \begin{pmatrix}11 & 12 & 13 & 14\end{pmatrix} & \begin{pmatrix}42 & 42 & 42 & 42\end{pmatrix}\end{pmatrix}.$$

All the inner vectors are of the same length. It is not hard to see that it is easy to flatten this vector of vectors to the following matrix:

$$\begin{pmatrix}1 & 2 & 3 & 4\\ 11 & 12 & 13 & 14\\ 42 & 42 & 42 & 42\end{pmatrix}.$$

This is a rectangular array with only scalar values as elements. Note that from a multidimensional array point of view there is no difference between the two arrays. This means that it is allowed to have nesting of arrays as long as the array can also be described as a multidimensional array of a higher rank with only scalar elements.

This raises the question of how to deal arrays where the elements are non-uniformly shaped arrays. Take the following vector of vectors for example:

$$\begin{pmatrix}\begin{pmatrix}1 & 2 & 3\end{pmatrix} & \begin{pmatrix}11 & 12 & 13 & 14\end{pmatrix} & \begin{pmatrix}42\end{pmatrix}\end{pmatrix}.$$

If this vector of vectors is converted into a matrix this matrix is no longer dense. One could argue that with some more complex description of the shape it would still be possible to describe this vector of vectors. However consider an example with truly multidimensional arrays.

$$\begin{pmatrix}\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix} & \begin{pmatrix}42 & 42 & 42\end{pmatrix}\\ \begin{pmatrix}11 & 22\end{pmatrix} & \begin{pmatrix}1 & 9\\ 2 & 8\\ 3 & 7\\ 4 & 6\end{pmatrix}\end{pmatrix}.$$

It is clear that it becomes non trivial to describe the shape of this array in an efficient way. This array does not qualify as a multidimensional array. We will call these arrays: irregular arrays.

It is not uncommon to have data patterns that are not rectangular in nature. Being able to deal with irregular arrays would allow programmers to write programs that more closely resemble the data pattern. This representation will most likely be more space efficient and could allow for speedup.

In SaC all arrays are multidimensional arrays. This allows the SaC compiler to generate efficient code. Rectangular arrays simplify the selection of elements, since selection on a rectangular array yields a rectangular array by definition. And it is easy to calculate the shape of the resulting array. This in turn means that often shape information can be inferred at compile time which allows for more efficient code.

The aim of this master project was to extend the SaC language and the SaC compiler to support irregular multidimensional arrays. In this chapter it is shown that it is non trivial how to deal with irregular arrays.

Of course there are several approaches one could take in order to get support for irregular arrays. In this thesis a few of these high level approaches are discussed. Of course some are more suitable then others. We will of course also discuss them in the context of SaC.

In order for programmers to use irregular arrays an extension to the SaC language will be introduced. This of course has to fit within the SaC type system and the SaC type hierarchy. It is undesirable to change the behavior of current SaC programs. This means that the introduced extensions will have to be non intrusive. One of the key constructs in the SaC language is the With-loop, see Section 2.3. It should of course be possible to use the With-loop in combination with irregular arrays.

The requirement of non intrusive extension applies to some extent to the run-time system. The current implementation of multidimensional arrays at run-time is efficient. And it is of course undesirable to lose performance. However, since the run-time representation is hidden from the user this will not break the semantics of the SaC language. Nonetheless, the aim is of course to provide a clean extension to the run-time system to facilitate easy maintenance.

1.1 Outline

The remainder of this thesis is organized as follows. In Chapter 2 the basics of the language will be introduced. Chapter 3 will discuss the run-time representation of a compiled SaC program. In Chapter 4 several approaches to irregular arrays are discussed. Chapter 5 provides the extension to the SaC language and the SaC run-time representation to support irregular arrays. This is followed by an evaluation of the implementation in Chapter 6. Chapter 7 will discuss related work. In Chapter 8 some possible future work is suggested. And finally in Chapter 9 we conclude if the implementation of irregular arrays in SaC is satisfactory.

CHAPTER 2

Single Assignment C

Single Assignment C [13] (or SaC for short) is a purely functional programming language with an ANSI-C like syntax. SaC is a true array language and employs multidimensional arrays as true first-class citizens.

This chapter serves as a minimal introduction to SaC to provide the reader with the required knowledge to understand the implementation of irregular arrays. For more general information about SaC see [1].

In Section 2.1 a short introduction to SaC is provided. Then a more extended explanation of arrays in SaC is given in Section 2.2. Following that the With-loop, which is one of the most important constructs in SaC, is introduced in Section 2.3. Once the arrays and the With-loop are introduced the SaC type hierarchy will be discussed in Section 2.4. Section 2.5 introduces the notion of user-defined types in SaC. For a more detailed description of SaC see [13, 23]

2.1 A Functional Subset of ANSI-C

As the name already suggests the syntax of SaC has a lot of similarities to ANSI-C [19]. However since SaC is a purely functional language it has to eliminate the elements of ANSI-C that can cause side-effects. This includes global variables and pointers. Also the notion of control flow is eliminated. This means that statements that influence this control flow, i.e. break, continue and goto, are not available in SaC. What remains is a purely functional language which can be mapped to an applied Lambda calculus [3]. One could see the core of SaC as a side-effect free variant of ANSI-C.

SaC is a typed functional language. Also SaC does eager evaluation. This means that all variables are place holders for values and can not contain unevaluated expressions.

While not all language constructs of core ANSI-C are available in SaC, it does offer some new language constructs that extend the language. One of the things, which is also found in C++ [25] and many other languages, is overloading. Overloading allows the programmer to specify multiple instances of a function. Because of the subtyping hierarchy of SaC (see Section 2.4) it is also possible to do overloading on subtypes. This allows for specialized functions for a given rank or shape.

As stated, there is no control flow in SaC and as a result of this there is only one return statement allowed per function. If a function has a return statement it has to be the last statement of that function. However in this return statement multiple return values are allowed. Again SaC is not the only language to allow return statements with multiple return values, this is for example also present in the Python [22] programming language. Also from a functional programming language perspective there is no reason to restrict functions to only yield one value.

2.2 Arrays in SAC

In SaC every value is considered to be an array. In SaC arrays are represented by two vectors. The *shape vector* and the *data vector*. First both vectors will be described here followed by some concrete examples of arrays in Section 2.2.1. The indexing of multidimensional arrays will be discussed in Section 2.2.2.

The *data vector* of an array contains all the elements of that array in row major order.

The *shape vector* defines the arrays structure (shape). The number of elements of the *shape vector* is the dimension, or rank, of the array. A vector has dimensionality one and thus its *shape vector* has one element, a matrix has dimensionality two and thus has a *shape vector* with two elements, etc. A special case are the scalar values, they have dimensionality zero which leads to an empty *shape vector*.

A *shape vector* $\begin{pmatrix} s_0 & .... & s_{n-1} \end{pmatrix}$ defines the structure of an n dimensional array which has s_i elements along its i^{th} axis. The associated *data vector* $\begin{pmatrix} d_0 & .... & d_{q-1} \end{pmatrix}$ must satisfy the equation $q = \prod_{i=0}^{n} s_i$. This constraint makes sure that the *data vector* can hold all the elements of the array.

All the elements of an array must be of the same data type. This is common in many languages. This design makes allocation easier and selection of elements from an array a lot simpler.

In SaC only dense arrays are supported. This means that all arrays are rectangularly shaped. This facilitates efficient selection of elements from an array. When using rectanguarly shaped arrays selection always yields an rectanguarly shaped array by definition. This in turn also makes it easier to infer the shape of the selected array.

Since SaC is a purely functional language defining an array means initializing all the elements of the array. Because there are no uninitialized values in an array selection always yields a valid new array with all values initialized. The downside of this is that if one would define an array of which a large part is never touched this part still has to be initialized.

2.2.1 Array Examples

In this section several examples of SaC arrays will be given. This should provide more insight into how multidimensional arrays are represented in SaC. The examples are limited to structures with dimensions that can still be visualized on a plane. But of course one can define multidimensional arrays of any dimension.

In Figure 2.1(a) one can see a 4-element vector. Since it is a vector, it by definition has a dimensionality of one, this results in a 1-element *shape vector*. The value of this one element is the number of elements in the vector (4 in this case). This then results in a *data vector* with 4 elements.

In Figure 2.1(b) a 2-dimensional array is displayed. This is more commonly known as a matrix. This results in a 2-element *shape vector*. Multiplying the elements from the *shape vector* gives the total number of elements in this array. Since this matrix has two rows and three columns the *data vector* has 6 elements. The *data vector* shows that the elements are stored in row major order.

$$\begin{pmatrix}1 & 2 & 3 & 4\end{pmatrix}$$

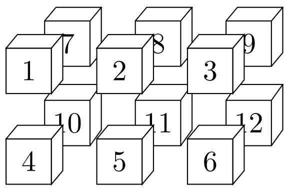
dimension:	1
shape vector:	[4]
data:	[1,2,3,4]

(a) A 1-dimensional array (vector) with 4 elements.

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

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

(b) A 2-dimensional array (matrix) with 2 rows and 3 columns (which adds up to 6 elements).



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

(c) A 3-dimensional array (tensor) consisting of 2 planes, 2 rows and 3 columns (a total of 12 elements).

42

dimension:	0
shape vector:	[]
data:	[42]

(d) A 0-dimensional array or a scalar, the data array holds 1 element by definition.

Figure 2.1: Examples of arrays in SaC.

The 3-dimensional array is the highest dimension where there is still a general agreement on how to visualize it on a 2-dimensional surface. However the example is very similar to the two previous examples. The resulting structure and its properties are available in Figure 2.1(c). As expected the number of dimensions is 3. This in turn means that the *shape vector* has 3 elements. Multiplication of the elements in the *shape vector* tells us that the *data vector* of the array has 12 ($2 \times 2 \times 3$) elements.

A special array case can be seen in Figure 2.1(d), a scalar. A scalar in SaC is a 0-dimensional array. This means that the *shape vector* of this array contains no elements, or in other words is empty. The *data vector* holds only 1 element. This is consistent with the formula for the size of the *data vector* since the neutral element of multiplication is 1.

2.2.2 Indexing

When dealing with multidimensional arrays it is important to be able to select elements from the array. In SaC this is done using the selection function. The selection function takes a selection vector and an array as arguments. The selection vector is used to index the array. The number of elements in the selection vector can be at most the dimension of the array. And naturally it is not allowed to select elements out of the bounds of the array. Subtracting the number of elements in the selection vector from the dimensionality of the array yields the dimensionality of the resulting array. In SaC indexing of arrays, like in ANSI-C, starts at zero. As an example matrix consider:

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

In Figure 2.2(a) the number of elements in the selection vector is the same as the dimension of the array. This means that the resulting array will have a dimensionality of 0, a scalar. In the case of Figure 2.2(a) the selection vector selects the second element from the second row. Which means that the selection in this case returns a scalar with the value 6.

In Figure 2.2(b) the selection vector only has 1 element while the argument array has a dimensionality of 2. This means that the result will have a dimensionality of 1, a vector. This is shown in Figure 2.2(b). The selection returns the vector $\begin{pmatrix} 5 & 6 & 7 & 8 \end{pmatrix}$.

The selection vector can also be the empty vector. This is selection of the entire argument array. This can be seen in Figure 2.2(c). This is in agreement with the expected dimensionality of the result of the selection. Since the selection vector has 0 elements and the argument array has a dimensionality of 2, the resulting array should also have a dimensionality of 2.

2.3 With-loop

One of the most important constructs in SaC is the With-loop. It defines the shape of an array and the values of the elements in that array. The With-loop is in essence a parallel map. This allows for efficient multithreaded execution [12, 11]. It is also possible to use reduction in combination with the With-loop.

The general syntax of the With-loop is

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

$$sel(\begin{pmatrix}1 & 1\end{pmatrix}, \begin{pmatrix}1 & 2 & 3 & 4\\5 & 6 & 7 & 8\end{pmatrix}) = 6$$

(a) Selection with a 2-element selection vector from a 2-dimensional array. This returns a 0-dimensional array (a scalar).

$$sel(\begin{pmatrix}1\end{pmatrix}, \begin{pmatrix}1 & 2 & 3 & 4\\5 & 6 & 7 & 8\end{pmatrix}) = \begin{pmatrix}5 & 6 & 7 & 8\end{pmatrix}$$

(b) Selection with a 1-element selection vector from a 2-dimensional array. This returns a 1-dimensional array (a vector).

$$sel(\begin{pmatrix}\,\end{pmatrix}, \begin{pmatrix}1 & 2 & 3 & 4\\5 & 6 & 7 & 8\end{pmatrix}) = \begin{pmatrix}1 & 2 & 3 & 4\\5 & 6 & 7 & 8\end{pmatrix}$$

(c) Selection with an empty selection vector from a 2-dimensional array. This selects the entire original 2-dimensional argument array.

Figure 2.2: Examples of the selection function in SAC.

Here, *lower_bound* and *upper_bound* are expressions that have to evaluate to integer vectors of equal length. Together they define a rectangular, generally multidimensional, index set. The identifier *idxvec* represents the elements of this set, this is similar to loop variables in a FOR-loop. Note that there is no order defined on the set. In SAC the specification of such an index set is called a *generator*. This *generator* can be associated with some potentially complex SAC expression. Thus a WITH-loop provides a mapping between index vectors and values, or in other words an array. As an example consider the WITH-LOOP

```
with {
  ( [0,0] <= iv < [4,5]) : 42;
}: genarray( [4,5], 0)
```

that defines a 4×5 matrix with all its elements set to 42. The scope of the *idxvec* (here named `iv`) is confined to the expression associated with the generator. It can be used in the expression of the WITH-loop. For example the WITH-loop

```
with {
  ( [0] <= iv < [4]) : iv[0];
}: genarray( [4], 0)
```

computes the vector $\begin{pmatrix}0 & 1 & 2 & 3\end{pmatrix}$. Note that here `iv` is a one element vector rather than a scalar. This requires us to select the first, and only, element from `iv` to get the desired result.

The generator does not actually define the shape of the resulting array. The shape of the resulting array is defined by the first expression following the key word `genarray`. Up till now those two have always coincided. But for example

```
with {
  ( [1] <= [i] < [4]) : i;
}: genarray( [5], 42)
```

computes the vector $\begin{pmatrix}42 & 1 & 2 & 3 & 42\end{pmatrix}$. This creates a five element vector, but only the three inner elements are defined in relation to the index set. All other elements are set to the *default value*, 42 in this case, which is given by the second expression following the key word `genarray`. This example provides a simplification considering the index variable. Enclosed in square brackets, `i` here is a scalar, which makes the selection of the first element obsolete.

WithLoop	$\rightarrow$	**with** { *[Body]* } : *Operator*
Body	$\rightarrow$	(*Generator*) : *Expr* ;
Generator	$\rightarrow$	(*Expr* | .) ($<$ | $\leq$) *IndexVars* ($<$ | $\leq$) (*Expr* | .)
IndexVars	$\rightarrow$	*Id*
	|	[*Id[,Id]**]
Operator	$\rightarrow$	**genarray** (*Expr, Expr*)
	|	**modarray** (*Expr*)
	|	**fold** (*Id, Expr*)
	|	**foldfix** (*Id, Expr, Expr*)

Figure 2.3: WITH-loop syntax.

A simplified syntax of the WITH-loop in EBNF [17] form is available in Figure 2.3. For more details about the WITH-loop see [12, 14]. The key word `genarray` is not the only key word that can be used in combination with the WITH-loop. The four most used key words are discussed below.

- **genarray**(*shp*, *default*): Generates an array with the shape as provide by *shp*. The index values that are covered by the generators of the WITH-loop will get the value in the matching expression. If multiple generators match an index the last defined generator is used. The index values that are not covered by the generators will be set to the value provided by *default*.
- **modarray**(*arr*): A new array with the same shape as the array *arr* is created. The index values that are covered by generators of the WITH-loop will get the value in the matching expression. Just like in the **genarray** case if multiple generators match an index the last defined generator is used. The index values that are not covered by the generators will be set to the value at that index in the array *arr*.
- **fold**(*fold_op*, *neutral*): This specifies a reduction on all the values produced by the generators. The *fold_op* function is a binary dyadic function which has *neutral* as its neutral element. As an example consider addition which has 0 as neutral element and multiplication has 1 as neutral element.
- **foldfix**(*fold_op*, *neutral*, *fixpoint*): This operation is very similar to the **fold** operation with one difference. Once the *fixpoint* is reached during folding the *With*-loop is terminated. A simple example is folding with multiplication. Once the intermediate result is zero all the coming multiplication will also be zero and there is no need to continue folding.

2.4 Type hierarchy

While all arrays in SaC are multidimensional arrays it is clear that the information available at compile time can range from full information, including the values of the array, to arrays of which we do not know anything, besides that it is an array of a given element type. SaC has three different kinds of array types. These are discussed below.

- **Array of Known Shape** (AKS): This is an array of which we know both the dimensionality as well as the shape. This means the number of elements is known at compile time.
- **Array of Known Dimension** (AKD): This is an array of which we do know the dimensionality but the shape of the array is unknown. As a result the total number of elements of the array is unknown at compile time.
- **Array of Unknown Dimension** (AUD): This is an array of which we have no other information besides that it is an array of a certain element type. The dimensionality is unknown which by definition gives us also an unknown shape and an unknown number of elements.

It is not hard to see that generated code for AKS arrays is more efficient than code generated for AUD arrays. On the other hand programmers probably want to write code at the AUD level to get nice generic code and let the compiler worry about generating efficient code. Of course the more generic code is written the more code reuse is possible.

The SaC compiler tries to infer shape information [23]. Since the more information is known at compile time the more boundary condition can be checked at compile time. This can lead to significant speedup.

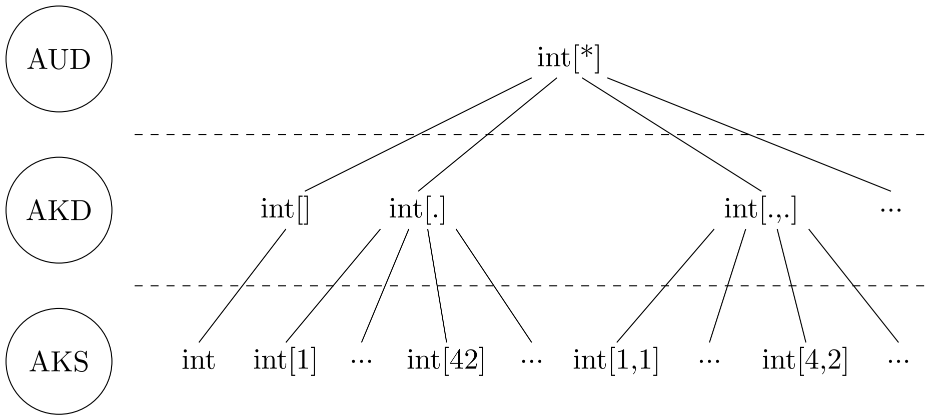


Figure 2.4: Type hierarchy for the integer type.

In Figure 2.4 the relation between the array types is visualized. Since the number of dimensions is not limited this type hierarchy is infinite, but the figure is a good illustration of the different types of arrays is SaC nonetheless. The figure also shows that the information available at the AKS level is indeed more than the information available at the AKD or AUD level.

Note the special case for the scalar values. For scalars the AKS and AKD type coincide. Since scalars have an empty *shape vector* there is no degree of freedom there.

Figure 2.4 shows that **int**[2] is a subtype of **int**[.]. And **int**[.] is a subtype of **int**[*]. In Figure 2.5 some SaC code is presented to show that the subtyping hierarchy really

```
bool foo(int[*] a)
{
  return( false);
}

bool foo(int[2] a)
{
  return( true);
}

int main()
{
  a = foo(42);                 // false
  b = foo([1,2]);              // true
  c = foo([ [4,5], [6,7] ]); // false

  return(0);
}
```

Figure 2.5: SaC code to illustrate overloading with respect to subtypes.

allows for overloading on rank and shape. In this case there are two functions `foo` with each only one argument. Both take an integer array. One is defined for an AUD array, while the other is defined for an AKS array, or more precisely for a two element vector. As is shown Figure 2.4, the most specialized version of a function will be used.

2.5 User-defined Types

In SaC there are a few built-in scalar types: int, bool, double, float and char. However even ANSI-C allows us to use the key word `typedef` to create "new" types. In ANSI-C this is only a type synonym. In SaC the key word `typedef` is also available but has a different meaning.

In SaC the key word `typedef` really introduces a new scalar type to the language. Because it is a new scalar type it also has its own type hierarchy. This allows the programmer to define arrays of the scalar type. A typedef in SaC is just one line of code:

```
typedef old_type new_scalar_type;
```

The *old_type* can be any type just as long as it is an AKS type, this includes scalar types. The *new_scalar_type* is just a string. This string will be the new scalar type to be defined. Of course all scalar types have to be unique within a program and the included libraries.

The type system in SaC is much more rigorous than the type system in ANSI-C. When a new scalar type is introduced it is not simply an alias. For this reason explicit casting is required. This casting really converts a from one type to another type. As an example consider the code in Figure 2.6. On line 6 a vector of two doubles is casted to the new type complex.

Because we are dealing with a new scalar type no functions are yet defined on it. Of

```
typedef double[2] complex;

int main()
{
  a = [4.0, 2.0];
  b = (:complex)a;

  return( 0);
}
```

Figure 2.6: Example SaC code of a definition a new type: complex. And the explicit casting of one type to another.

course, on arrays of the new type we can do all the array operators like select, reshape etc. If the programmer wants to use some function, like addition, on the new scalar type he or she has to implement this function first.

CHAPTER 3

Run-time representation

The SaC compiler compiles SaC code to ANSI-C. This ANSI-C code is in turn compiled to machine code using an ANSI-C compiler. This ANSI-C compiler can be chosen by the programmer.

3.1 Arrays

In SaC all values are arrays. And as shown in Section 2.2 SaC supports truly multidimensional arrays. At run-time a SaC array is represented by two ANSI-C arrays. The *descriptor* and the *data array*.

The *data array* is a direct mapping of the *data vector* of a SaC array. The *data array* holds, just like the *data vector*, all the values of the SaC array in row major order.

3.1.1 Descriptor

The *descriptor* describes the shape of a SaC array but also contains information that is required at run-time. A *descriptor* is implemented as an ANSI-C integer array. It has at least 3 elements.

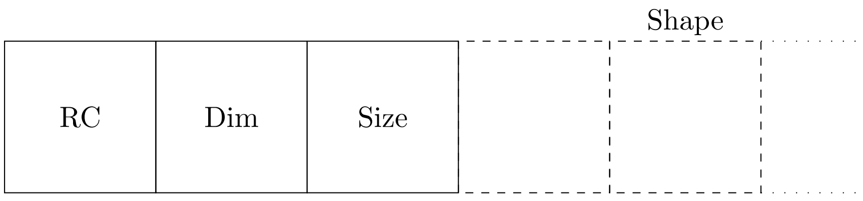


Figure 3.1: A *descriptor* belonging to the run-time representation of a SaC array.

In Figure 3.1 one can see the 3 base elements of the descriptor. The `shape` of an array is appended to this. This `shape` part of the *descriptor* can be seen as a direct representation of the *shape vector* of a SaC array.

The `Dim` field of the descriptor hold the dimension, or rank, of the array. This value tells us the number of shape elements in the *descriptor*.

The `Size` field holds the size of the *data array*. This is the product of the elements in the *shape vector*.

At run-time it is important to keep track of the number of references to an array. Once an array is no longer referenced it can be deallocated. The `RC` field provides the counter of the number of references to that array.

3.1.2 Scalars

In Section 2.2 it is shown that in the SaC language even scalar values are arrays. However at run-time representing scalars as arrays is not very efficient. Especially since there is already support for scalars in ANSI-C. Scalars are kept on the stack instead of on the heap, this makes reference counting unnecessary. And since the dimension and shape of a scalar are known by definition there is no need for a *descriptor*. This saves space at run-time.

3.2 User-defined Types

In Section 2.5 the concept of user-defined types on the SaC language level is explained. While on the language level a new scalar type is introduced this is not possible at run-time. But by definition any user-defined type can be mapped down one of platform scalar types.

The reason that on the language level of SaC it is only possible to define a new user-defined type on AKS types has to do with the run-time representation of user-defined types. Because a user-defined type is an AKS type of another scalar type, possibly of another user-defined type. But in the end a user-defined type can be represented as an AKS array of a scalar type native to the platform.

Consider the following typedef:

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

If one would construct a matrix of 10×10 elements, and the elements are of type complex. It is not hard to see that this matrix can be flattened out to be a tensor of $10 \times 10 \times 2$ doubles.

This flattering of user-defined types makes them at run-time just SaC arrays. The user-defined types are resolved rather early in the SaC compiler which allows all the optimizations that work on arrays to by definition also work on user-defined types, or arrays of user-defined types.

CHAPTER 4

Design Space of Irregular Arrays

This chapter gives an overview of techniques that can be used to implement irregular arrays in general. For each technique we will discuss a motivation why, or why not, the approach might be useful in general, and more specifically in the context of SaC.

Remember that we defined an irregular array to be an array with a non-rectangular shape. As an example remember the irregular array

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} & \begin{pmatrix} 42 & 42 & 42 \end{pmatrix} \\ \begin{pmatrix} 11 & 22 \end{pmatrix} & \begin{pmatrix} 1 & 9 \\ 2 & 8 \\ 3 & 7 \\ 4 & 6 \end{pmatrix} \end{pmatrix}.$$

It is unclear what the rank of this array is and describing the shape also becomes non trivial.

The outline of this Chapter is as follows. First a very simple padding approach will be discussed in Section 4.1. In Section 4.2 an early approach for irregular arrays, the bit vector, will be discussed. Followed by the nested shape array approach in Section 4.3. The final section of this chapter, 4.4, introduces an approach using the nesting of arrays.

4.1 Padding

The easiest solution to irregular arrays seems to just try to catch this into a rectangular array. Take the following vector of non-uniformly shaped vectors

$$\begin{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} & \begin{pmatrix} 2 \end{pmatrix} & \begin{pmatrix} 3 & 4 & 5 & 6 \end{pmatrix} & \begin{pmatrix} 7 & 8 & 9 \end{pmatrix} \end{pmatrix}.$$

One could convert this into a rectangular matrix. However to be rectangular all the rows have to be the same length. Since the longest row has 4 elements all the other rows have to be padded to also become 4 elements long. This would result in the matrix

$$\begin{pmatrix} 0 & 1 & \emptyset & \emptyset \\ 2 & \emptyset & \emptyset & \emptyset \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & \emptyset \end{pmatrix}.$$

This directly shows that this approach can be very space inefficient. It only takes one large partition and many small partitions to blow up the required space significantly compared to the actual used space.

Also the set of numerical types does not contain the Ø element. And one of the requirements of an array is that all the elements are of the same type. One could of course take 0 as the empty element but this generates invalid results when for example doing reduction with multiplication.

The problem is generic in the sense that there is no special numerical value that could represent the empty element. If we would want to add 1 to all the elements of the array, it does not matter what is chosen as the empty element. Since plus 1 is valid on all numerical values the element will be updates and no longer be the empty element.

This approach seems like an easy solution to solve the issue of irregular arrays. However, as explained in this section padding is not a feasible solution.

4.2 Bit vector

One of the early suggested approaches to irregular arrays, as suggested in [24], is to have a data vector and a bit vector that will have the same number of elements. This bit vector will be one 1 on the elements that define the start of a new vector and 0 on all the other elements.

This is a very simple approach to have nested vectors and it is not that hard to implement this in a language itself. Of course to protect the user from a major book-keeping effort it makes sense to make it a language feature so that it is hidden from the user. However, behind the syntactic sugar the implantation will roughly be the same.

$$\begin{array}{rl}
\text{vector of vectors:} & \big((0\ \ 1)\ \ (2)\ \ (3\ \ 4\ \ 5\ \ 6)\ \ (7\ \ 8\ \ 9)\big) \\
\text{data vector:} & \big(0\ \ 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\big) \\
\text{bit vector:} & \big(1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 0\big)
\end{array}$$

Figure 4.1: The desired vector of irregular shaped vectors. The data vector and the corresponding bit vector.

Figure 4.1 provides an example of how to represent a vector of vectors using one data vector and one bit vector.

As shown the approach will require another vector that has the same number of elements as the total number of elements of all the vectors that are to be nested. While this second vector will of course require extra space it can be implemented efficiently using a bit vector. Depending on the base type of the original array a bit vector can be an order of magnitude smaller than the original vector.

An important thing to consider is the performance impact this approach can have when selecting an element. Since it is unknown where the next nested vector starts the bit vector has to be scanned sequentially, and in the worst case scanned entirely. This moves the complexity of selection from $O(1)$ to $O(n)$.

Also this approach was not designed for more than 1 level of nesting. If one tries to do deeper nesting operations like selection will become even less efficient.

The use of bit vectors provides a relatively simple solutions to the problem of irregular arrays. It could be easily implemented in the SaC language with two vectors and the With-loop. One issue is the increased complexity of selection from $O(1)$ to $O(n)$.

But the main reason for rejecting this approach is that it is unable to deal with truly multidimensional arrays. And since SaC is all about truly multidimensional arrays, this approach does not fit.

4.3 Nested shape array

In order to get nested arrays a solution might be to keep a flat *data vector* around and just have a nested *shape vector*. This reduces the problems complexity from multidimensional arrays to vectors.

The main problem with this approach is that it introduces a bootstrap problem since then we will still need support for multilevel irregular vectors to support the nested *shape vector*.

There is also of course the question how one would express the dimensions of a nested array in this way. For example some elements could have a dimension of one, while others have a different dimension.

Consider the following vector with irregular elements

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} & \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} & \begin{pmatrix} 42 & 42 \\ 42 & 42 \\ 42 & 42 \end{pmatrix} \end{pmatrix}.$$

The shape of this vector or even the rank are not trivial to determine. The rank is non trivial because the first element of the vector has a rank of 1 but the other two elements have a rank of two. It is obvious that it is not possible to describe the shape in a single vector. If described as a vector of vectors the shape could look like

$$\begin{pmatrix} 3 & \begin{pmatrix} 2 & 2 \end{pmatrix} & \begin{pmatrix} 3 & 2 \end{pmatrix} \end{pmatrix}.$$

In essence, the shape is no longer described by a vector but by a tree.

A strong advantage of this approach is that the *data vector*, just as in regular arrays, contains all the data. One can benefit from this since we have strong locality in the data as a result of the implementation.

However, depending on the implementation the one *data vector* can also be a disadvantage because there is no longer a guarantee that the replacement of an element will be done by an element of the same shape. This is illustrated in the following pseudocode:

```
d    = [ [0, 1], [2], [3, 4, 5, 6], [7, 8, 9] ];
d[1] = [ 11, 12 ];
```

In this approach also the complexity of selection increases. Since in order to calculate the offset into the *data vector* of an element one needs to know the total number of element in front of the desired element to be selected. It is easy to see that it is not possible to calculate this efficiently. Of course we also need to calculate the total number of elements we want to select. This is for the same reason no longer trivial to calculate efficiently.

As described in Chapter 3 the *shape vector* of an array is at run-time part of the *descriptor*. However switching to nested shape vectors would require non trivial modification in the *descriptor*. This could result in performance loss when dealing with regular arrays. This combined with the increased complexity of selection makes this approach not the best suitable candidate for the implementation of irregular arrays in SaC.

4.4 Nested Arrays

A more dynamic setting for irregular arrays is to nest the arrays. The “outer” array of an irregular array will then just hold references to other arrays. We call this approach nested arrays. When using nested arrays there is no longer one flat *data vector*. This approach is in some aspects similar to the approach used in the Qube programming language [27, 26].

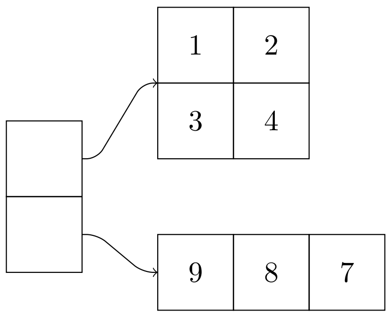


Figure 4.2: A vector with references to other arrays.

In Figure 4.2 the nested array approach is visualized. It is clear that this approach introduces some overhead since the references have to be stored as well. However the required extra space only grows linearly with the number of elements in the outer vector.

An advantage of this approach is that it allows for rectangular arrays on both levels. Only the indirection of the references separates and thus prevents direct selection of elements.

Another advantage of this approach is that the general array structure will not have to be changed radically, just the element types change. The good thing about this is that people who are familiar with multidimensional arrays will not have much problem understanding the structure of the nested array.

However, there is of course the downside of performance impact. Since we are using references the obtained locations in memory might be on opposite ends of the address space. This could mean that while the data feels local it is not. This could have a major performance impact.

Another performance penalty is paid when doing selection. In Figure 4.2 this means that first an element in the outer vector is selected. Then the reference is followed to the new array and then one can do selection on that array.

With this approach you also have to pay a penalty for selection. This is unavoidable with irregular arrays. However with the nested array approach the penalty is only paid when a reference is followed. In the other cases regular arrays can be used. This approach does allow for truly multidimensional array on all levels. This makes the approach the best candidate for implementation in SaC.

CHAPTER 5

Implementation of Nested Arrays

In Chapter 4 the choice for the implementation of irregular arrays with nested arrays is motivated. In this chapter we will discuss the implementation of nested arrays in SaC. The extension to the language will be discussed in Section 5.1. This is followed by a discussion about the actual run-time implementation in Section 5.2.

5.1 Language

In Chapter 2 a short introduction the SaC language is provided. The chapter explains about arrays in SaC. The two major requirements for arrays are that an array is rectangularly shaped and that all the elements of an array are of the same type.

It is obvious that the requirements of all arrays to be rectangular prevented programmers from writing irregular arrays in SaC. This is solved by the nested array approach trough the use of references. In Figure 5.1 an example of this is provided. The three "inner" arrays `B`,`C` and `D` are all rectangularly shaped. The same holds for the outer array `A`. The array `A` does not contain the arrays `B`, `C` and `D` but only references to those arrays. This means that the requirement of arrays to be rectangularly shaped still holds with the nested array approach.

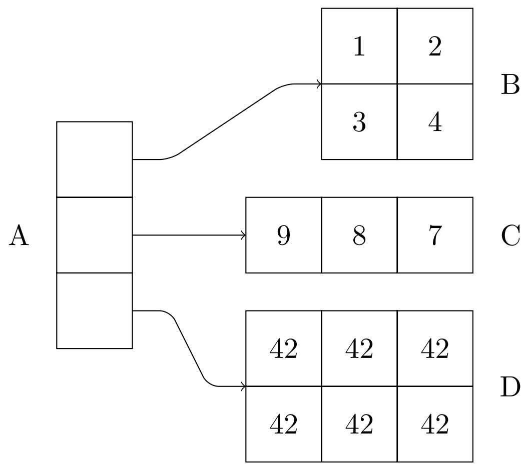


Figure 5.1: Irregular array with an outer vector containing pointers to three other arrays.

The other requirement is that all the elements of an array have to be of the same type. If one considers the example provided in Figure 5.1 it is easy to see that this is the case for the arrays B, C and D. The problem arises when determining the element type of the array A.

To find out what kind of elements the array A contains one must consider the type hierarchy of SaC, as explained in Section 2.4. This tells us that the type of array B is **int**[2,2], the type of array C is **int**[3] and the type of array D is **int**[2,3]. This would mean that the elements of the array A would be **int**[2,2], **int**[3] and **int**[2,3]. These are three different types.

As explained in Section 2.4, SaC has subtyping. This subtyping hierarchy provides us with a solution. The three provided types are all AKS. Now lets consider the AKD types of the three arrays B, C and D. This gives us **int**[.,.] for array B, **int**[.] for array C and **int**[.,.] for array D. This shows us that now array B and array D are of the same type. However there is still no one type for the array A. For this we again need to go one level up. All three arrays B, C and D are subtypes of the AUD type **int**[*].

This means that if we could define a new scalar type that somehow contains **int**[*] types it could be possible to construct the array A with those scalar types. This is very similar to a user-defined type.

However as stated in Section 2.5 and then explained in Section 3.2 it is in SaC only allowed to have user-defined types of AKS types. Since the implementation of the existing user-defined types is very efficient and rather straight forward it would be undesirable to implement those user-defined types with nested arrays. For this reason an extension to the typedef key word is provided.

```
typedef nested old_type new_scalar_type;
```

Adding the extra nested key word now allows the programmer to define a typedef with non-AKS types as the *old_type*. Note that it does allow AKS types so a programmer could choose to use nested arrays with AKS types.

5.1.1 Enclosing and disclosing

With the extension to the typedef key word it is now possible to define new scalar types that can contain non-AKS arrays. However in order to make proper use of this the programmer will need some way to convert from an array to the new scalar type. And of course the same is required for the reverse operation.

We will call this enclosing and disclosing. Enclose takes an array and encloses it to a scalar value. Disclose does the reverse. It takes a scalar value and extracts an array from it.

It is insufficient to just provide two functions, enclose and disclose. This is a result of the type system of SaC. Since one could easily define two nested types that have the same base type. Consider the following code:

```
typedef nested int[.] row;
typedef nested int[.] row2;
```

Now if there was just one enclose function there is no way for the type system to infer to what new scalar type to enclose an integer vector. For this reason the SaC compiler introduces an enclose and a disclose function for each nested type.

If one would define a new nested type foo then the disclose function will be called *enclose_*foo and the disclose function is called *disclose_*foo. These are the functions the programmer has to use. Since these functions are non ambiguous, the type system can infer the resulting type. If one would define

```
typedef nested int[.] row;
```

this will yield the two functions as provided in Figure 5.2.

```
row enclose_row(int[.] from);
int[.] disclose_row(row from);
```

Figure 5.2: Automatically generated functions as a result of the line:
typedef nested int[.] int_vec.

5.2 Run-time representation

In the previous section the notion of enclose and disclose is introduced. It is clear what these function do on an abstract level but it is not trivial how to implement this at run-time. This Section will introduce two possible ways to implement nested arrays at run-time and a motivation is provided why the choice of implementation is in favor of one of them.

5.2.1 Representation as a Structure

It seems logical to put the pointer to the *descriptor* and the pointer to the *data array* into an ANSI-C structure. This would nicely fit the abstract concept of enclose. The new scalar would just be the structure. Disclose could just take this structure and return the *descriptor* and the *data array*.

However, as explained in Section 3.1, reference counting is done on all arrays. But when converting to a structure we would need to reference count this structure as well. This extra level of reference counting will introduce additional overhead, which is of course undesirable.

```
typedef nested int[.] row;

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

  a = enclose_row(data);
  b = [a, a];

  return(0);
}
```

Figure 5.3: A vector with two times the same element which is an enclosed 3-element integer vector.

Consider the example in Figure 5.3. The array *b* contains two references to *a*. If we would not reference count *a*, it would be impossible to know when *a* could be deallocated. And since *a* holds a reference to the array *d* it is also unclear when the array *d* can be deallocated.

This approach would require the implementation of a new level of reference counting to the run-time system. This is not desirable since this would introduce more overhead to the memory management system, which in turn results in a performance penalty.

5.2.2 Two element void pointer array

One can work around the extra level of reference counting by using a 2-element ANSI-C array of void pointers. The first element of this ANSI-C array holds the pointer to the *descriptor* and the second element will hold the pointer to the *data array.*

If one would store the two element ANSI-C array of void pointers on the stack the need to reference count that ANSI-C array disappears.

There is of course still the requirement to properly reference count the array that is enclosed. However since the descriptor to that array is just stored in the two element ANSI-C array of void pointers, it is possible to just do the reference counting directly on the enclosed array.

If one would now declare an array of the new nested type, this is similar to a rectangular array. The *descriptor* is the same as for any array. However the *data array* is slightly modified since we will need to store two pointers. This means the *data array* holds twice as many elements. This in turn also means that operation on arrays, like selection etc., have to be modified to also allows for arrays with nested elements.

This approach seems to be more in tone with the existing implementation of arrays in SaC. The reference counting mechanism does not have to be modified which makes implementation easier. For arrays of nested types there only is a modification required to the *data array* which is easier to do than modifying the *descriptor.* For the reasons listed above this is the approach that we chose to implement nested arrays in SaC.

CHAPTER 6

Evaluation

In Chapter 5 the implementation of nested arrays in SaC is discussed. Now that the design decisions and implementation have been discussed this chapter will evaluate those decisions and show cases where nested arrays can be very useful and of course also some cases where nested arrays will not be useful.

All the measurements are performed on a Sun/Oracle Fire X4440 server. The sever is fitted with four 4-core AMD Third-Generation Opteron 8356 processors running at 2.3 GHz. The amount of RAM installed is 16 gigabyte. The system runs CentOS 5.5 64-bit Linux, the running kernel is 2.6.18-164.11.1.el5. On this system an integer is 4 bytes, a double is 8 bytes and a pointer is also 8 bytes.

In Section 6.1 it will be shown that it is now possible to have nested arrays. Section 6.2 will show the penalty for using nested arrays on problems that have rectangular shaped data. Matrix multiplication is used as an example. In Section 6.3 an example is given where the use of nested arrays can lead to a significant drop in memory requirements as well as an increase in performance. Section 6.4 shows that it is now possible to have sparse matrices in SaC.

6.1 Arrays of strings

This section provides a qualitative evaluation. We show that it is now possible to have irregular arrays in SaC programs. Consider the two string ”Hello” and ”World!”. One can easily represent them as as two character arrays.

$$\begin{pmatrix} 'H' & 'e' & 'l' & 'l' & 'o' \end{pmatrix} \text{ and } \begin{pmatrix} 'W' & 'o' & 'r' & 'l' & 'd' & '!' \end{pmatrix}$$

Figure 6.1 shows that it is possible to create a vector of two nested elements. In the example a vector is constructed where the first element contains the array representing ”Hello” and the second element contains the array representing ”World!”.

It is important to note that even though one can now create such a nested array the operations on arrays, like selection, are defined in the SaC standard library. Which means that those functions are not available by default. However, one can of course choose to overload those functions.

While this is a rather simple example, it immediately shows that the implementation of nested arrays in SaC now allows the programmer to write programs that were not possible to write before.

```
typedef nested char[.] string;

int main()
{
  hello  = ['H', 'e', 'l', 'l', 'o'];
  hello2 = enclose_string(hello);
  world  = ['W', 'o', 'r', 'l', 'd', '!'];
  world2 = enclose_string(world);

  hello_world = [hello2, world2];

  return(0);
}
```

Figure 6.1: Vector of "strings". The strings are enclosed character vectors.

6.2 Matrix multiplication

To illustrate the overhead introduced by nested arrays we will use matrix multiplication. Matrix multiplication is chosen since it is relevant numeric kernel that is often used for benchmarking and is also often used in applications.

The comparison in this chapter is done with two square $n \times n$ matrices. Of course one could easily do matrix multiplication of $n \times m$ matrices.

In Section 6.2.1 the default SAC implementation of matrix multiplication, so without nested arrays, is discussed. Section 6.2.2 covers a matrix multiplication implementation with nested rows of the matrix. In Section 6.2.3 the extreme nested array case is considered where all the elements of the matrix are 1-element nested vectors.

For all there approaches the run-time for different matrix sizes will be presented as well as the spatial requirements. A comparison of the three approaches is done in Section 6.2.4.

6.2.1 No nested arrays

The SAC code for matrix multiplication without nested arrays is provided in Appendix A. It is directly clear that there is heavy usage of functions declared in the standard library. Matrix multiplication is done as is to be expected. The second matrix is transposed and then row wise multiplication is performed.

The formula for the required space, in bytes, of this approach for an $n \times n$ matrix is:

$$f(n) = n^2 \times 8 + 7 \times 4 = 8n^2 + 28$$

The formula is rather straight forward. There are n^2 doubles in the *data array* and 7 integers in the *descriptor*.

6.2.2 Nested rows

One of the implementations that uses nested arrays for matrix multiplication could be to make all the rows of a matrix to be a nested vectors. There then would be one outer vector that holds all these rows. The SAC code for this is available in Appendix B.

The formula for the required space of this approach, in bytes, is:

$$f(n) = n \times (n \times 8 + 6 \times 4) + 2n \times 8 + 6 \times 4 = 8n^2 + 40n + 24$$

Each row is represented by a nested vector. Each of these nested vectors requires n doubles in the *data array* and a *descriptor* of 6 integers. The outer vector has $2n$ void pointers in its *data array* and also has a *descriptor* of 6 integers.

6.2.3 Nested elements

To quantify and measure the overhead introduced by nested arrays we also show an extreme example. It is unlikely that anyone will write code as presented here. Having said that, the approach itself still works and provides a way to describe a worst case scenario.

This extreme approach is to make each element a 1-element nested vector. All these new scalars are then stored in an $n \times n$ matrix. This results in a large amount of 1-element vectors. Which is a huge burden on the memory system. The SAC code for this approach can be found in Appendix C.

The required space, in bytes, for a $n \times n$ matrix of nested 1-element vectors is provided by the following formula:

$$f(n) = n^2 \times (1 \times 8 + 6 \times 4) + (2n^2) \times 8 + 7 \times 4 = 512n^2 + 28$$

Each 1-element vector requires a *data array* with 1 double. Since it is a vector there is also the need of *descriptor* of 6 integers. The outer matrix will have a *descriptor* of 7 integers and a *data array* that has to hold $2n^2$ void pointers.

6.2.4 Comparison

When looking at the formulas for the different approaches it is clear that the nested array approach introduce overhead in this case. The relative space requirement of the nested array approaches compared to the matrix multiplication that does not use nested arrays is plotted in Figure 6.2.

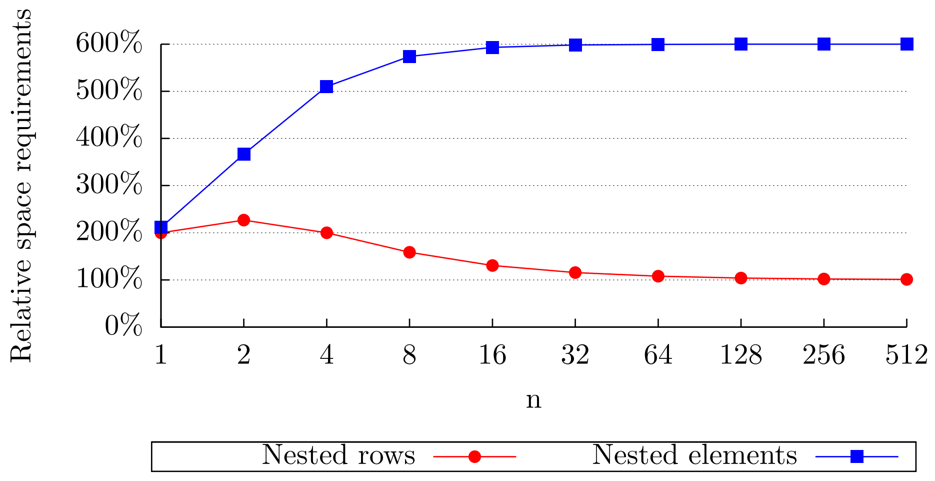


Figure 6.2: Relative spatial requirements of matrix an $n \times n$ matrix in the nested vector approach and in the nested element approach compared to the approach without any nested arrays.

It is clear from the figure that the space requirements of the nested row approach are not that far off from the implementation without nested arrays. When comparing the formula this is also to be expected. This shows that the spatial overhead of the nested row approach is only $40n$, since this factor is linear in n it becomes less of an issue for large values of n. As a result the space requirements for the nested row approach asymptotically approach the space requirements of the implementation without nested arrays.

For the nested element approach the overhead is a lot larger. Figure 6.2 shows that for large values of n the nested element approach takes close to 6 times as much space compared to the approach without nested arrays. While this is a constant factor, this can easily become an issue for large matrices.

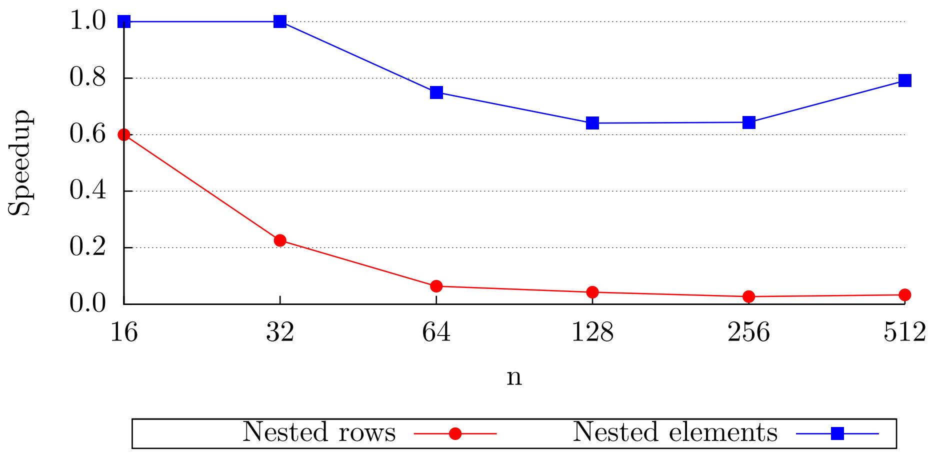


Figure 6.3: Speedup of matrix multiplication with two $n \times n$ matrices for the nested row approach and the nested element approach compared to the approach without nested arrays.

In Figure 6.3 a plot is shown of the speedup of the nested vector approach and of the nested element approach compared to the approach without any nested arrays. For small values of n the measurement are most likely not very accurate.

The nested row approach is slower than the approach without any nested arrays but still does rather well considering the introduced overhead. The nested element approach on the other hand is an order of magnitude slower. This is to be expected since the introduced memory overhead of that approach is huge. This further stresses the point that no one should, and probably also no one would, write code like that of the nested elements.

6.3 Triangular shaped data

Another possible data pattern is that in the shape of a triangle. Of course there are many possible variations on this data patterns but for this example the simple case will be considered. This means that the first row has one element, the second row has two elements etc. In the example it is assumed that the elements of the triangle are integers.

6.3.1 With a matrix

In the traditional case often a matrix is constructed. If there are n rows then the matrix will have n by n integers. Of course only in the last rows there are n useful elements.

The required space, in bytes, for a triangle with n rows is provided by the following formula:

$$f(n) = n^2 \times 4 + 7 \times 4 = 4n^2 + 28$$

This is just an $n \times n$ matrix of integers. Since it is a matrix, the *descriptor* contains 7 integers.

6.3.2 With nested arrays

One can easily use nested arrays to hold triangular shaped data. Each row is just a vector of integers. These vectors are enclosed. The new scalar values are all be put in a vector.

Each vector also has a descriptor, this introduce overhead for each vector. Of course this overhead becomes less of an issue once the number of elements in the vector increase.

The required size, in bytes, of a triangle with n rows with the nested array approach is provided by the following function:

$$f(n) = \frac{n \times (n+1)}{2} \times 4 + n \times (6 \times 4) + 2n \times 8 + 6 \times 4 = 2n^2 + 42n + 24$$

This formula requires some explanation. The inner vectors contain in total $\frac{n\times(n+1)}{2}$ integers. Each inner vector has a descriptor of 6 integers. The outer vector contains $2n$ void pointers and also has a *descriptor* of 6 integers.

```
typedef nested int[.] row;

int[.] vec(int s)
{
  res = with {
            ( . <= iv <= . ) : s;
          } : genarray([s], 0);

  return(res);
}

int main()
{
  pyramid = with {
                ( . <= [i] <= . ) : enclose_row(vec(i + 1));
              } : genarray([128], enclose_row([]));

  return(0);
}
```

Figure 6.4: Example SAC code to generate triangular shaped data.

To show that it is not hard to write the triangle with nested arrays see the code example in Figure 6.4. This is a triangle with one element on the first row, two elements on the second row etc. The values of each row are not very interesting, in the example but this can of course be made as complex or as non trivial as one would like.

6.3.3 Comparison

Even without the formulas for the spatial requirements of the two approaches it is not hard to see that the approach with nested arrays for large values of n will require approximately half the space of the approach without nested arrays.

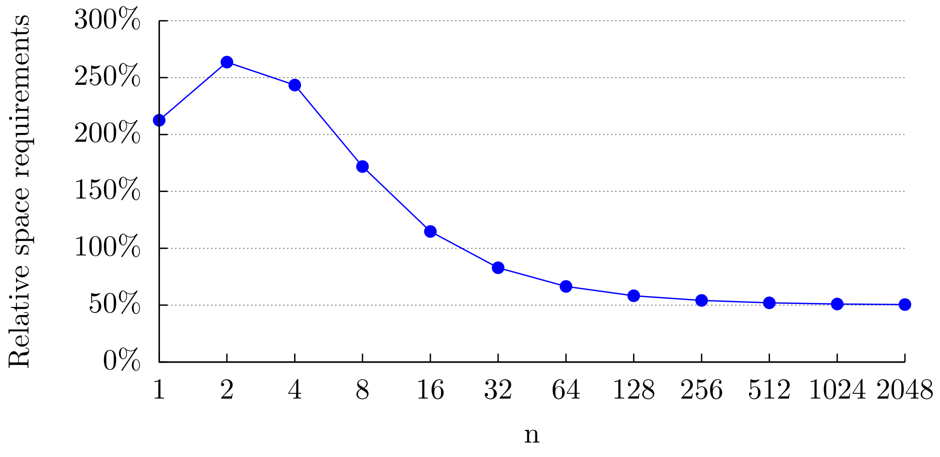


Figure 6.5: Relative space required for a triangular shape data of n rows of the nested array approach compared to the approach without nested arrays.

Figure 6.5 shows that for small values of n the approach with nested arrays takes up more space but quickly becomes more space efficient. To determine where the nested array approach becomes more space efficient one has to solve the following equation for n:

$$4n^2 + 28 = 2n^2 + 42n + 24$$

This yields two solutions. $\frac{21-\sqrt{422}}{2}$ which is smaller then 1 and thus is not useful here. And $\frac{21+\sqrt{422}}{2}$ which is approximately 20.4, this tells us that once a triangle has 21 or more elements the nested array approach is more space efficient.

Since SAC is a functional language defining an array means initializing all the elements of this array. Keeping this in mind lets consider a program where we have triangular shaped data and we want to add 1 to each of its elements.

In the case without nested arrays this would mean that for the elements in the triangle the addition is performed and the other elements are set to zero. Whereas the approach with nested arrays does not have unused elements.

Figure 6.6 shows the speedup of the use of nested arrays. For small values of n the measurements are probably not that reliable. However when n gets larger the speedup gets close to 2.

This example clearly shows that for some problems the use of nested arrays can be more space efficient and can even lead to better performance.

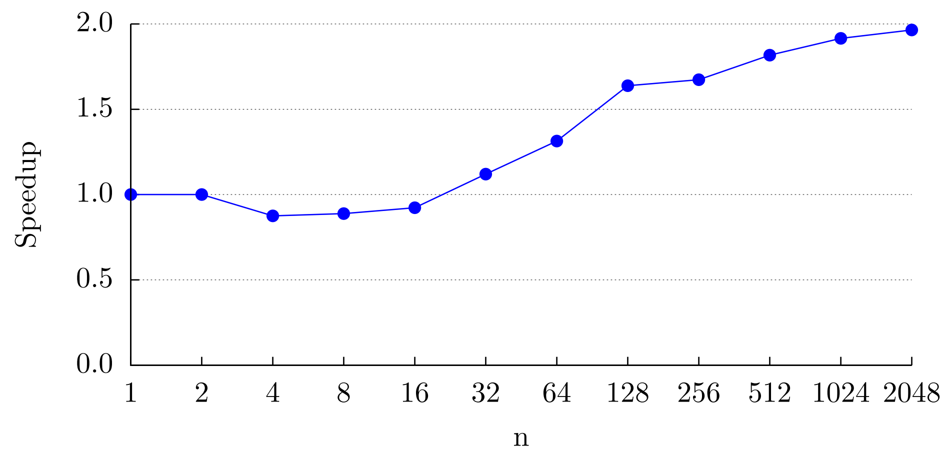


Figure 6.6: Speedup of adding one to the elements of triangular shaped data of the nested array approach compared to the approach without nested arrays.

6.4 Sparse matrices

A sparse matrix is a matrix that is populated mainly with zeros. In general this means that storing a sparse matrix as a dense matrix is not very space efficient. For large matrices it might not even be possible to hold the dense matrix in memory when only keeping the non zero elements in memory would be no problem.

One can consider a sparse matrix as a vector of sparse vectors. One can represent such a sparse vector in an efficient way by just using two vectors. One vector holds all the indices of non-zero elements and the other vector holds the values corresponding to those indices. Since all the other elements are zero this would remove the requirement to store the entire vector. Each row of the sparse matrix can contain a different number of non zero elements. Thus, storing all the sparse vectors in one vector requires irregular arrays.

Consider the following matrix:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 6 & 7 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

This matrix could be represented by the following two irregular arrays.

$$\begin{pmatrix} \begin{pmatrix} 8 \end{pmatrix} \\ \begin{pmatrix} \; \end{pmatrix} \\ \begin{pmatrix} 1 & 6 \end{pmatrix} \\ \begin{pmatrix} \; \end{pmatrix} \\ \begin{pmatrix} 4 \end{pmatrix} \\ \begin{pmatrix} 2 & 3 & 4 \end{pmatrix} \end{pmatrix} \text{ and } \begin{pmatrix} \begin{pmatrix} 1 \end{pmatrix} \\ \begin{pmatrix} \; \end{pmatrix} \\ \begin{pmatrix} 2 & 3 \end{pmatrix} \\ \begin{pmatrix} \; \end{pmatrix} \\ \begin{pmatrix} 4 \end{pmatrix} \\ \begin{pmatrix} 5 & 6 & 7 \end{pmatrix} \end{pmatrix}.$$

It is not hard to see that this is representation of the matrix is more space efficient than just storing the entire dense matrix. This ratio becomes even large when the number of non-zero elements get smaller. Note that it is important for this approach to be able to represent the empty vector.

For the experiments in this section we will use sparse matrices that that have a the following pattern. Consider a square $n \times n$ matrix that has a 1 on the top left to bottom right diagonal and has a 2 on the top right to bottom left diagonal. This is unless the index of the column is a multiply of 5. As an example consider the matrix for $n = 8$.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

It is clear that this is a sparse matrix. If we would multiply this matrix with a vector it would of course yield another vector. However for a dense matrix this would mean doing just regular matrix-vector multiplication. But in the sparse case we can only multiple the non-zero elements since the other multiplication will by definition yield zero.

```
typedef nested int[.] row;

inline
int[.] mul (row[.] A, row[.] B, int[.] C)
{
  res = with {
          ( . <= [i] <= .) {
            idx = disclose_row(A[i]);
            val = disclose_row(B[i]);
          } : with {
            ( 0 * shape(idx) <= [j] < shape(idx) ) :
              C[idx[j]] * val[j];
          } : fold(+, 0);
        } : genarray( shape(C), 0);

  return(res);
}
```

Figure 6.7: Example SaC code that shows how to multiply a sparse matrix represented by two nested arrays with a vector. A holds the indices, B holds to values belonging to those indices and C is the vector.

In Figure 6.7 the SaC code to multiply a sparse matrix with a vector is shown. While the code is slightly more complex than that of normal matrix-vector multiplication it is still easy to read and not that hard to understand.

To calculate the required space for this matrix using nested arrays we make a simplification. We assume that all the rows contain 2 non-zero elements. This yields the following formula for the spatial requirements, in bytes, of a dense matrix using nested arrays.

$$f(n) = 2 \times (n \times (2 \times 4 + 6 \times 4) + 6 \times 4 + (2n \times 8)) = 96n + 48$$

Each row has a *data array* of 2 integers and a *descriptor* of 6 integers. There are n of such rows. All the rows are put in a vector which has a descriptor of 6 integers and a *data vector* containing $2n$ void pointers. We of course need two of such arrays, one to store the indices and one to store the values at those indices.

The required space for a dense matrix, in bytes, is provided by the following formula:

$$f(n) = n^2 \times 4 + 7 \times 4 = 4n^2 + 28$$

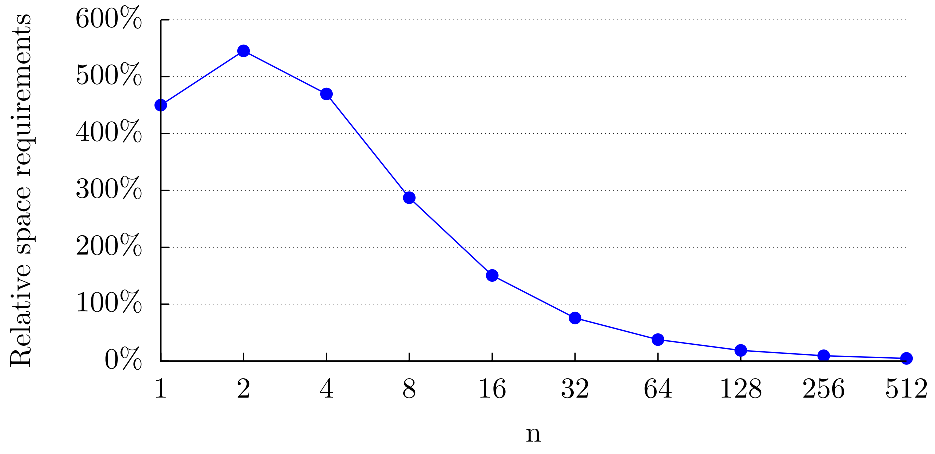


Figure 6.8: Relative space requirements of an $n \times n$ matrix of the sparse representation using nested arrays compared to the dense representation.

In Figure 6.8 the relative space requirements of the sparse matrix representation using nested arrays compared to a dense matrix representation is plotted. Again for small values of n the overhead of the nested arrays is clearly visible. But, for larger values of n the dense matrix representation using nested arrays becomes an order of magnitude smaller.

The obtained results of the measurements of doing matrix-vector multiplication are plotted in Figure 6.9. Since the number of elements in the matrix grows quadratically but the number of non-zero elements only grows linearly the speedup increases as the matrix gets larger. The figure clearly shows that as the percentage of non-zero elements becomes smaller the speedup increases. This is to be expected.

Sparse matrices are not uncommon. This examples illustrates a possible implementation of sparse matrices in SAC using nested arrays. This implementation allows for a significant reduction in spatial requirements while at the same time providing a large increase in performance.

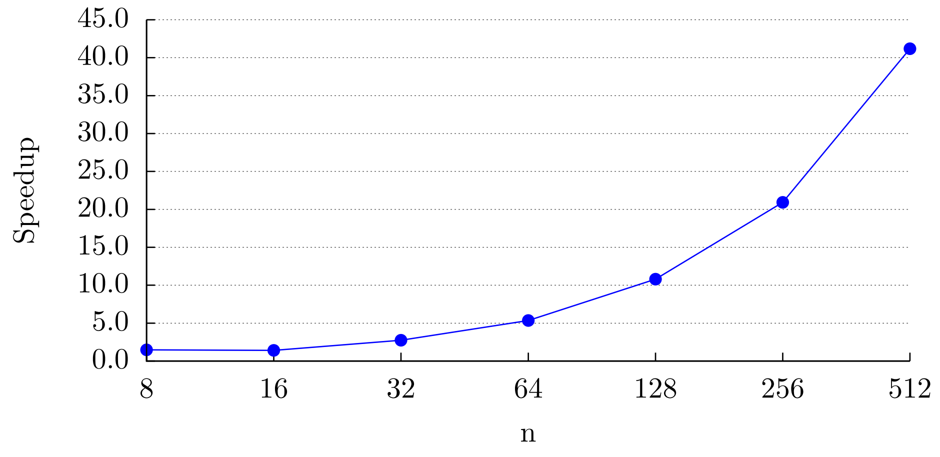


Figure 6.9: Speedup of multiplication of a $n \times n$ sparse matrix with a n-element vector of the sparse matrix representation with nested arrays compared to the dense matrix approach.

.

CHAPTER 7

Related work

It is not hard to imagine that SaC is not the only language where one would like to have a mechanism to deal with irregular arrays. This chapter will discuss some other languages that have a way of dealing with irregular arrays. In these discussions a small comparison to SaC will be provided.

7.1 Nesl

Nesl [5] is a data-parallel language. It is a strongly-typed strict first-order functional language. Nesl uses a sequence as primitive parallel data type. Parallelism is achieved exclusively trough operations on these sequences [6].

A sequence in Nesl can be compared to a vector in SaC. In Nesl it is allowed to have a sequence of sequences. This is in a way support for irregular arrays. Nesl does not support truly multidimensional arrays. Which in turn results in that there is no support for irregular multidimensional arrays.

As is described in [4] Nesl flattens out nested sequences. This results in one *data array* and a number of so called "sedges" (for segment descriptor) to describe the tree like structure of the nesting. This is comparable to the nested shape array from Section 4.3.

Consider the following vector of vectors:

$$\begin{pmatrix}\begin{pmatrix}2 & 1\end{pmatrix} & \begin{pmatrix}7 & 0 & 3\end{pmatrix} & \begin{pmatrix}4\end{pmatrix}\end{pmatrix}.$$

In Nesl this is represented by three vectors. One is the *data vector*, called value, and two so segdes.

$$\begin{array}{rl} \text{segdes1:} & \begin{pmatrix}3\end{pmatrix} \\ \text{segdes2:} & \begin{pmatrix}2 & 3 & 1\end{pmatrix} \\ \text{value:} & \begin{pmatrix}2 & 1 & 7 & 0 & 3 & 4\end{pmatrix} \end{array}$$

7.2 Haskell

Haskell [21] is extended to support data parallelism by Nepal [8] and Data Parallel Haskell [7]. The underlying techniques are mostly based on Nesl. It makes sense to compare Data Parallel Haskell to SaC since SaC has built in support for data parallelism trough the With-loop.

In order to be able to use nested parallel arrays in Haskell they first have to be flattened. This is similar to the approach used in Nesl.

In Haskell there is no real support for multidimensional arrays. It is possible to have a nesting of vectors to simulate this. Some libraries in Haskell present multidimensional arrays to the users but they are implemented as vectors of vectors.

Since Haskell uses deferred garbage collection it is not possible to do in place updates of data. This is possible in SaC. The nested array implementation does not change this behavior in SaC.

The different approaches in dealing with nested multidimensional arrays will most likely result in different performance for different sets of applications. If one would consider an application where the data only has to be flattened once the Haskell approach would probably be faster. However if one would have a program that requires modifications to the nested multidimensional array the SaC approach is most likely more efficient.

7.3 APL2

APL2 [9] is the successor to APL [18]. APL is an interactive array-oriented language. It has support for multidimensional arrays.

One of the primary enhancements of APL2 is the support for nested arrays. The abstract notion of nested arrays in SaC and APL2 is rather similar. Enclose and disclose operator were introduced in APL2 and served as a starting point for the implementation of nested arrays in SaC.

However there are fundamental differences between APL and SaC. SaC is a typed language and APL is not. Also APL is an interpreted language while SaC is a compiled language.

Another important distinction is that in APL lazy evaluation can be used [2], this means that expressions are not evaluated until they are needed. SaC on the other hand always does eager evaluation.

7.4 FORTRAN

One of the most widely used array languages is FORTRAN [16]. It is an procedural, imperative programming language. FORTRAN is an array programming language and has support for multidimensional arrays.

A lot of numerical benchmarks have a FORTRAN implementation that usually score very well. FORTRAN is a language that is often used in high performance computing.

However, it is not possible to have irregular arrays in FORTRAN. This means that all the programs have to use rectangular data. This shows that in this extend SaC is more flexible.

CHAPTER 8

Future Work

In this chapter some possible future work with respect to nested arrays in SaC will be discussed.

8.1 Array Functions

Once the programmer defines a nested type, the compiler automatically generates the *enclose* and *disclose* functions for that type. However, this is not the case for functions like selection, rotation etc. These could be automatically generated and would make the life of a programmer probably a lot easier.

This does introduce a new problem. Right now there are only a few built-in functions in SaC. For example the selection function as explained in Section 2.2.2 is not a built-in function, but a function defined in the SaC standard library. And since at compile time of the standard library only the built-in types are known, it is only possible to compile the functions for those types.

This has been an issue for a long time in SaC. However with user-defined types the problem does not occur because of the way they are handled by the SaC compiler. But, with irregular arrays it is now possible to really introduce new types to SaC, these new types can not be flattened to built-in types by the compiler, and thus require new functions.

A possible solution might be to only define a skeleton for the functions like selection, rotate etc. The compiler could then infer, from the used types in a program and the used functions in a program, which skeleton function to convert, and compile, into “real” functions for that program.

8.2 Elimination of operations

Right now most of the nested array operations are hidden from the optimizations. This is of course undesirable. It might often be the case that an array is enclosed just to be disclosed in the next operation. Consider the code in Figure 8.1, this does not look to bad. However, if the compiler inlines the function calls, as in Figure 8.2, the chance for optimization become even more obvious.

It is not hard to see that there is no real point in enclosing an array only to disclose it again. If the compiler could optimize this a part of the overhead introduced by nested arrays would already be omitted.

```
import StdIO: all;

typedef nested int[.] row;

inline
row compute(row a)
{
  c1 = disclose_row(a);

  /* Some computation */
  c2 = ....

  res = enclose_row(c2);

  return(res);
}

inline
void print(row a)
{
  p = disclose_row(a);

  print(p);
}

int main() {
  a = enclose_row([0,1,2]);
  b = compute(a);
  print(b);

  return(0);
}
```

Figure 8.1: Example of SAC code with nested arrays.

```
import StdIO: all;

typedef nested int[.] row;

int main() {
  a = enclose_row([0,1,2]);

  /* b = compute(a) */
  c1 = disclose_row(a);

  /* Some computation */
  c2 = ....

  b = enclose_row(c2);

  /* print(b) */
  p = disclose_row(b);
  print(p);

  return(0);
}
```

Figure 8.2: Example of SaC code with nested arrays with inlined function calls.

8.3 Regular shaped data analysis

A major advantage of SaC is the ability to abstract away from the rank and the shape of arrays. A useful extension to this would be if one could also abstract away from the fact that arrays are irregular or not.

It is discussed that implementation of user-defined types is more efficient than the implementation of nested arrays. If the compiler would detect that an nested arrays is used, and all the elements in this nested array are of the same AKS type, the compiler could then use a user-defined type instead.

Take complex numbers for example. One can describe a complex number by the AKS type **double**[2]. However this **double**[2] is a subtype of **double**[.]. Now there is choice to use the AKS variant and flatten out the complex numbers or to use the nested arrays. While nested arrays offer more flexibility that flexibility might not be required by the user. If the compiler could detect that all the instances of the nested type complex are actually of the AKS type **double**[2] it would choose to use the user-defined types.

CHAPTER 9

Conclusions

An irregular array is an array where the elements of this array are not uniformly shaped. In most array languages arrays are flattened. However, if the elements of an array are not uniformly shaped this is no longer trivial to do.

We have discussed a few high level approaches to irregular arrays. It is not hard to see that when using irregular arrays a penalty has to be paid. For example selection is no longer trivial and is by definition more complex than when using rectangular arrays.

Nested arrays were chosen as best suitable approach for SaC. This required modifications to run-time representation of a SaC program. The general concept is that an array can contain references to other arrays. This allows the arrays themselves to remain rectangular.

The approach chosen to implement these irregular arrays in SaC is on a conceptual level similar to nested arrays in APL2. We "enclose" arrays to scalar values which allows for rectangular arrays to contain these enclosed arrays.

The implementation of nested arrays requires the programmer to explicitly call the enclose and disclose functions. This can be seen as a disadvantage. However, this does make it clear to the programmer that he or she is not working with rectangular arrays.

We have shown that indeed a penalty is paid when using nested arrays in SaC for problems that are rectangular in nature. This is shown for matrix multiplication in Section 6.2.

We have also shown that when the data is triangularly shaped the required space is approximately half of that of when using a matrix. This is shown in Section 6.3. That section also shows that we obtained a speedup of almost 2 when using irregular arrays for triangularly shaped data.

It is now also possible to describe sparse matrices in SaC using irregular arrays. The use of sparse matrices allows for a significant reduction in the spatial requirements while at the same time allowing substantial speedups. Section 6.4 shows that for a 512×512 element sparse matrix we get a speedup of approximately 41.

These examples show that if a program contains irregular data the use of irregular arrays can lead to both significant reductions in spatial requirements as well as substantial speedups. At the same time irregular arrays allow the programmer to describe the data in a way that more closely resembles the actual shape of the data.

Bibliography

[1] Single Assignment C homepage. `http://www.sac-home.org`.

[2] Philip Samuel Abrams. *An APL Machine*. PhD thesis, Stanford, CA, USA, 1970. AAI7022146.

[3] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[4] Guy Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 21:102–111, 1994.

[5] Guy Blelloch and Parallel Ram Model. Nesl: A Nested Data-Parallel Language. Technical report, 1990.

[6] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[7] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon P. Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM.

[8] Manuel M.T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal – Nested Data-Parallelism in Haskell. In *IN EURO-PAR 01*, pages 524–534. Springer-Verlag, 2001.

[9] International Business Machines Corporation. *APL2 Programming: Language Reference*. IBM, 1988.

[10] Adin D. Falkoff and Kenneth E. Iverson. The Design of APL. *IBM Journal of Research and Development*, 17(5):324–334, 1973.

[11] Clemens Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[12] Clemens Grelck and Sven-Bodo Scholz. SAC — From High-level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters*, 13(3):401–412, 2003.

[13] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[14] Stephan Herhut, Sven-Bodo Scholz, and Clemens Grelck. Controllling Chaos — On Safe Side-Effects in Data-Parallel Operations. *ACM SIGPLAN Notices*, 44(5):9–10, 2009.

[15] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, 21:359–411, September 1989.

[16] IEC. *ISO/IEC 1539-1 (1997-12): Information technology — Programming languages — Fortran — Part 1: Base language.* 1997.

[17] International Standard: Information technology – Syntactic metalanguage – Extended BNF. ISO/IEC 14977: 1996(E), First Edition, December 1996.

[18] Kenneth E. Iverson. A Programming Language. *Managing Requirements Knowledge, International Workshop on*, 0:345, 1962.

[19] Brian W. Kernighan. *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[20] C. B. Moler, J. Little, and S. Bangert. *PRO-MATLAB User's Guide.* The MathWorks, Inc., Sherborn, MA, 1987.

[21] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[22] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[23] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[24] Bob Smith. A programming technique for non-rectangular data. *SIGAPL APL Quote Quad*, 9:362–369, May 1979.

[25] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[26] Kai Trojahner. *QUBE — Array Programming with Dependent Types.* PhD thesis, University of Lübeck, Institute of Software Technology and Programming Languages, Lübeck, Germany, 2011.

[27] Kai Trojahner and Clemens Grelck. Dependently Typed Array Programs Don't Go Wrong. *Journal of Logic and Algebraic Programming*, 78(7):643 – 664, 2009.

APPENDIX A

Matrix multiplication SaC code

```
#ifndef SIZE
#define SIZE 512
#endif

use Array:all;
use StdIO:all;

inline double[.,.] matmul( double[.,.] A, double[.,.] B)
{
  BT = transpose( B);
  C = with {
        ( . <= [i,j] <= .) : sum( A[[i]] * BT[[j]]);
      }: genarray( [ shape(A)[[0]], shape(B)[[1]] ], 0d);
  return( C);
}

int main()
{
  A = genarray( [SIZE,SIZE], 1.0);
  B = genarray( [SIZE,SIZE], 1.0);
  C = matmul( A, B);

  printf( "C[0,0] = %f\n", C[0,0]);
  return(0);
}
```

APPENDIX B

Matrix multiplication with nested rows SaC code

```
#ifndef SIZE
#define SIZE 512
#endif

import Array:all;
use StdIO:all;

typedef nested double[.] row;

/*
 * Function to selection an element from a vector with
 * row elements.
 */
inline
row[*] sel( int[.] idx, row[*] array)
{
  new_shape = _drop_SxV_( _sel_VxA_( [0], _shape_A_(idx)),
                          _shape_A_(array));
  res = with {
          ( . <= iv <= . ) {
            new_idx = _cat_VxV_( idx, iv);
          } : _sel_VxA_(new_idx, array);
        } : genarray( new_shape, enclose_row([]) );
  return( res);
}

/*
 * Shorthand selection function
 */
inline
row[*] sel( int idx, row[*] A)
{
  return( sel( [idx], A));
}

/*
 * Redefinition of the infix multiplication of two
 * row variables
 */
inline
double[.] (*) (row A, row B)
{
  res = disclose_row(A) * disclose_row(B);

  return(res);
}

/*
 * Implentation of transpose for matrices of nested rows
 */
inline
row[.] transpose(row[.] A)
{
  B =  with {
          (. <= i <= .) : enclose_row(
            with {
              (. <= j <= .) : disclose_row(A[j])[i];
            } : genarray( [SIZE], 0.0)
          );
       } : genarray( [SIZE], enclose_row([0.0]));

  return(B);
}

inline
row[.] matmul( row[.] A, row[.] B)
{
  BT = transpose(B);

  C = with {
        (. <= i <= .) : enclose_row(
          with {
            ( . <= j <= . ) : sum(A[i] * BT[j]);
          } : genarray( [SIZE], 0.0)
        );
      } : genarray( [SIZE], enclose_row([1.0]));

  return (C);
}

int main()
```

```
{
  e = with {
      } : genarray( [SIZE], 2.0);

  A = with {
      } : genarray( [SIZE], enclose_row(e));
  B = with {
      } : genarray( [SIZE], enclose_row(e));

  C = matmul(A, B);

  /* We have to print something or all the calculations
   * are optimized away */
  print(disclose_row(C[0])[0]);

  return(0);
}
```

APPENDIX C

Matrix multiplication with nested elements SaC code

```
#ifndef SIZE
#define SIZE 512
#endif

import Array:all;
use StdIO:all;

typedef nested double[.] element;

/*
 * Function to selection an element from a vector with
 * element elements.
 */
inline
element[*] sel( int[.] idx, element[*] array)
{
  new_shape = _drop_SxV_( _sel_VxA_( [0], _shape_A_(idx)),
                          _shape_A_(array));
  res = with {
          ( . <= iv <= . ) {
            new_idx = _cat_VxV_( idx, iv);
          } : _sel_VxA_(new_idx, array);
        } : genarray( new_shape, enclose_element([0.0]) );
  return( res);
}

/*
 * Shorthand selection function
 */
inline
element[*] sel( int idx, element[*] A)
{
  return( sel( [idx], A));
}

/*
 * Redefinition of the infix operation for multiplication
 * of two nested element elements.
 */
inline
element (*) (element A, element B)
{
  res = enclose_element(
          disclose_element(A) * disclose_element(B)
        );

  return(res);
}

/*
 * Redefinition of the infix operation for multiplication
 * of two nested element vectors.
 */
inline
element[.] ((***)) (element[.] A, element[.] B)
{
  res = with{
          (. <= iv <= .) : A[iv] * B[iv];
        } : genarray( _shape_A_(A), enclose_element([3.0]));

  return(res);
}

/*
 * Sum all the elements of a element vector
 */
inline
element sum(element[.] A)
{
  res = with {
          (0*_shape_A_(A) <= iv < _shape_A_(A)) :
            disclose_element(A[iv]);
        } : fold(+, 0.0);

  res = enclose_element(res);

  return(res);
}

/*
 * Transpose a matrix of nested element elements.
 */
inline
element[+] transpose( element[+] A)
{
  res = with {
          ( . <= iv <= . ) : A[reverse(iv)];
        } : genarray(reverse(_shape_A_(A)),
                     enclose_element([2.0]));
  return( res);
}


inline
element[.,.] matmul( element[.,.] A, element[.,.] B)
{
  BT = transpose( B);

  C = with {
        ( . <= [i,j] <= .) : sum(A[i] * BT[j]);
      } : genarray( [_shape_A_(A)[[0]], _shape_A_(B)[[1]]],
                    enclose_element([1.0]));

  return (C);
}



int main()
{
  A = with {
      } : genarray( [SIZE,SIZE], enclose_element([2.0]));
  B = with {
      } : genarray( [SIZE,SIZE], enclose_element([2.0]));

  C = matmul(A, B);

  print(disclose_element(C[0,0]));

  return(0);
}
```