A Compiler Backend for Generic Programming with Arrays

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Dietmar Kreye

Kiel
Juni 2003

1. Gutachter:	Prof. Dr. Werner Kluge
2. Gutachter:	Prof. Dr. Alex Shafarenko

Datum der mündlichen Prüfung:	9. Juli 2003

Berichte aus der Informatik

Dietmar Kreye

A Compiler Backend for Generic Programming with Arrays

Shaker Verlag
Aachen 2004

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the internet at http://dnb.ddb.de.

Zugl.: Kiel, Univ., Diss., 2003

Printed in Germany.

ISBN 3-8322-2538-2
ISSN 0945-0807

Shaker Verlag GmbH • P. O. BOX 101818 • D-52018 Aachen
Phone: 0049/2407/9596-0 • Telefax: 0049/2407/9596-9
Internet: www.shaker.de • eMail: info@shaker.de

Contents

List of Figures

List of Symbols

$\mathcal{T}_{\mathrm{AUD}}$ 17

Set of all types with unknown dimension and unknown extent (AUD $\equiv$ array of unknown dimension), e. g., `int[*]` $\in \mathcal{T}_{\mathrm{AUD}}$.

Chapter 1

Introduction

Scientific computing — sometimes referred to as number crunching — is one of the most important and successful genres of computer science, used for instance in statistics, in finance, or for the simulation of physical, chemical and industrial processes. Typically, these applications involve fairly complex, non-uniform operations on large matrices/arrays. In traditional problems, such as aerodynamics or weather forecast, these arrays represent two- or three-dimensional spaces. However, modern applications, e. g., quantum mechanics or finance, require also higher dimensional arrays.

Scientific application programs usually require extremely long execution times and operate on huge amounts of data. Therefore, a programming language suitable for such applications should provide the means to specify code with a high efficiency both in terms of time and memory demand. In order to complete the computation in a certain time frame, it is often inevitable to employ multiple processors which simultaneously work on the solution of a given problem, e. g., the weather forecast for the following day. Hence, languages with support for concurrent program execution are most desirable. Other criteria for the choice of a programming language are the expressive power with respect to array operations, and the potential for code reuse as well as for maintenance of existing programs. Such features facilitate less error-prone program development which is of particular importance since in modern society computers are often used for applications which for safety and security reasons have a high demand for program correctness.

The current state of affairs is that the number crunching scene is almost exclusively dominated by imperative programming languages like FORTRAN-77 [ANSI78] or C [KR88]. Their low level of abstraction allows experienced programmers to achieve utmost performance, but program development and maintenance are time-consuming and error-prone. Implementing array operations requires problem-specific loop-nestings. Whenever the size or dimension of the arrays involved changes, the loop boundaries or the loop nestings must be modified as well. It is often rather difficult to make sure that all array accesses are legal and that no boundary violations will occur at runtime. Moreover, these languages handle arrays as references rather than values, i. e., array modifications are performed as destructive updates irrespective of the number of references that exist to the array. If an algorithm needs multiple instances of an array, the array must be copied explicitly. In C, arrays could be allocated and removed dynamically but it is the programmer's duty to organize these operations correctly. As a consequence, programs written by unexperienced programmers often suffer from memory leaks or dangling array pointers. Furthermore, traditional imperative languages are not very suitable for concurrent program execution. Due to the state-based semantics, programs are inherently sequential, complex program analyses are therefore required to identify program parts which can be executed in any chosen order without altering the meaning of the program. Unfortunately, many features of imperative languages, e. g., call-by-reference parameters, implicit side-effects via global variables, pointers in C, or `common` blocks in FORTRAN, make this analysis extremely difficult and inefficient.

Some derivatives of FORTRAN — FORTRAN-90 [ABM$^+$92] and HPF [HPFF97] — address these problems by extending FORTRAN-77 by a large set of built-in array operations which are applicable to arrays of any size and dimension, e. g., element-wise extensions of scalar operations. While this allows for a more concise, less error-prone program specification and facilitates concurrent program execution within the boundaries of a single array operation, this approach does not eliminate the fundamental drawbacks of imperative languages. Besides, code becomes less generic if operations have to be applied to subsets of array elements only. Although regularly structured cases are addressed by the so-called triple notation, a step back to loops and scalar operations is often inevitable. In either case, the code must be tailor-made for a concrete dimension. Moreover, these languages provide no means of defining customized abstractions which are of similar generality as the built-in primitives.

The approach taken by functional programming languages avoids many of the shortcomings known from the imperative world. Their semantics is based on the principle of context-free substitutions [CF58, Bare84, HS86, Hank94] rather than on a stepwise modification of states. As a consequence, functional programs are free of side-effects which considerably facilitates high-level code optimizations. Moreover, the Church-Rosser-Property of the functional paradigm guarantees determinacy of results irrespective of execution orders, hence, functional languages are well-suited for implicit concurrent program execution. Besides, many typical features of these languages [FH88, BW88, Bird98, Read89], like implicit memory management, polymorphism, higher-order functions, partial function applications, and lazy evaluation, allow for a higher level of abstraction in program specifications. However, most functional languages have only limited support for array processing.

At least HASKELL [Peyt03] and CLEAN [PE01a, PE01b] provide a sound integration of arrays into the functional domain [Gron97]. Besides primitive array operations, like selection or arithmetics, they offer so-called array comprehensions which are used to map scalar operations onto specified index intervals of arrays. But unfortunately, the support is restricted to arrays of fixed dimension only. Even worse, the absence of side-effects causes considerable problems with regard to the efficient implementation of array operations. Conceptually, operations must consume their argument arrays and create new result arrays, rather than overwriting existing ones, which is generally very costly both in terms of time and memory demand [HB85].

Some languages like ML [MTH90] and its derivative OCAML [Lero02] circumvent this problem by implementing arrays as state-full data structures and by providing side-effecting operations on them. Although this approach considerably improves runtime performance, it sacrifices almost all the benefits of the functional paradigm as far as arrays are involved, which in turn brings about all the difficulties known from imperative languages.

A completely different approach is taken by the functional programming language SISAL [MSA$^+$85]. It does without most of the functional frills, as for example polymorphism, higher-order functions, partial applications, and lazy evaluation, since these features could introduce considerable slowdowns. It provides an implicit memory management for arrays based on reference counting [Cohe81, Cann89, FO95], which allows to implement array operations destructively whenever possible. Furthermore, its call-by-value semantics is exploited for sophisticated code optimizations [Cann89, CE95] and for the implicit generation of concurrently executable code [SSM88, HB93, PAM93]. As a result, the SISAL compiler [Cann93] generates code which outperforms equivalent FORTRAN programs in a multiprocessor environment [OCA86, Cann92].

However, SISAL does not offer substantial advantages in terms of programming techniques. As in imperative languages, the programmer is still asked to specify array operations as iteration loops whose index ranges must be adapted to array shapes. Moreover, SISAL supports one-dimensional arrays only. Higher-dimensional arrays must be represented by nestings of such arrays, which de-grades performance with growing dimension. Some of these shortcomings are addressed in more recent versions of SISAL, i. e., SISAL 2.0 [BCOF91, Olde92] and SISAL-90 [FMSD95, FO95] — however, none of them have ever been implemented.

More general support for arrays is provided by array-oriented programming languages such as APL [ISO84, ISO93], J [Burk96, Bern93], K [Kx98], and NIAL [JJ93]. Originating from a mathematical notation for arrays [Iver62], the main objective in the design of these languages is to offer means for specifying algorithms on arrays in a very concise and abstract manner. They typically have a call-by-value semantics with implicit memory management and give support for *shape-invariant programming*, i. e., all operations/functions can be defined in a generic way that allows arguments to have arbitrary dimension and size.

This *generic approach* of programming with arrays has a lot of benefits. Being able to define new shape-invariant array operations allows the programmer to adjust the set of primitive array operations to the needs of any given algorithm. Instead of problem-specific loop nestings, new and more generally applicable operations may be defined, which subsequently may be combined to express the desired functionality. As a consequence, programs become more modular and easier to understand, which in turn increases code reusability and makes program development less error-prone. However, overloading array operations with many different combinations of array shapes, including scalars, causes difficulties with respect to runtime efficiency, since it usually requires dynamic typing and execution in an interpreting environment. Although some optimization techniques have been invented [Brow85] and various attempts have been made to compile such programs [DO86, Budd88, Bern97], runtime efficiency is in many cases less than satisfactory.

SAC (short for Single Assignment C) is a more recent development of a functional array processing language [Scho96, Scho03]. It is designed to combine the advantages of APL and SISAL: Although SAC provides means to specify truly shape-invariant array operations, the SAC compiler manages to generate code whose runtime performance is competitive to those of high-performance imperative languages such as FORTRAN-90 or HPF. SAC offers only a small set of built-in array operations, more complex operations may be defined by means of the so-called `with`-loop — a SAC-specific array comprehension. In contrast to the `for`-loop in SISAL, the `with`-loop allows to define operations which completely abstract from the shapes of the arrays involved. Due to the expressive power of the `with`-loop all primitive array operations known from other languages like APL or FORTRAN-90 can be implemented in SAC itself. Placed into SAC libraries, these operations may serve as building blocks for real world applications, making their definitions more concise, less error-prone, and more comprehensible.

Typically, such program specifications introduce many intermediate arrays. In order to achieve competitive runtimes, powerful optimization techniques are required that avoid the actual creation of these intermediate arrays whenever possible. While in an imperative setting this task turns out to be difficult, it can be readily done in SAC by applying so-called `with`-loop folding techniques [Scho98b]. In [Scho98a, GS00] it is shown for several program examples that `with`-loop folding can eliminate large numbers of intermediate arrays. In fact, the resulting code is almost identical to what can be accomplished for programs that directly implement the desired functionality in an element-wise manner rather than benefiting from an APL-like programming style.

Prerequisite for the success of `with`-loop folding, as well as many other high-level code optimizations which have been invented for SAC, is the so-called *static shape inference*, which tries to infer the shapes of all arrays used in a program. With knowledge of the shapes involved, it is even possible to pre-evaluate some array operations at compile time (*partial evaluation*).

Unfortunately, inferring shapes statically is impossible in certain situations, e. g., if external library functions are compiled separately or input data have unknown shapes. Even worse, whenever a recursive function is applied to an argument whose shape is changing with each recursive call, shape inference may be undecidable. For the time being, the SAC compiler rules out all programs for which static shape inference fails. As a consequence, programs written in SAC must be recompiled whenever the shape of input data changes, generic library functions can not be compiled separately, and implementing certain algorithms requires some awkward code design or is even impossible.

Another language which has been designed with static shape inference in mind is FISH [Jay98, Jay99]. FISH is a higher order, polymorphic, ALGOL-like functional language for array processing. The FISH compiler manages to create code whose runtime demand is two orders of magnitude smaller than the demand of equivalent HASKELL programs and two to four times smaller than the demand of equivalent OCAML programs. But, similar to SAC, the expressive power of the language is restricted significantly to ensure that the shape of every array can be determined statically.

The aim of this thesis is to describe a new compiler backend that is based on static shape inference but uses a general approach to eliminate the shortcomings of recent compilers: In order to get utmost performance, shape-specific code is generated whenever exact shapes can be inferred statically. However, to preserve the full expressive power of the language, more generic code is created if static shape inference fails. The language of choice for this new compiler backend will be SAC.

The basic idea of this approach is to make use of a hierarchy of array types with different levels of shape information. The most specific array types specify an exact shape whereas more general types prescribe an exact dimension only or contain no shape information at all. During the compilation process programs are typed as shape-specific as possible and subsequently the hierarchy of array types is translated into a corresponding hierarchy of array representations.

Implementing such a hierarchy of array types is burdened with two major problems. The first one is about resolution of function overloading. One of the key features of SAC is the support for function overloading with respect to shapes, i. e., SAC programs might contain shape-specific as well as non-shape-specific instances of a single function. The semantics of SAC prescribes that for each function application the most specific instance suitable for the arguments must be used. If the compiler is capable of inferring all array shapes, resolving this function overloading is trivial. Knowing the argument shapes of an application, it is statically decidable which instance of the function has to be used. But if static shape inference fails, the compiler must generate additional code for the function application which dynamically chooses the matching instance at runtime.

The second problem arises in the code generation phase of the compiler. The compiler must generate code for a hierarchy of array types with different levels of shape information. For this purpose, appropriate data structures must be found to represent these shape informations. Runtime evaluations reveal that it is more efficient to map the hierarchy of types to a corresponding hierarchy of representations, instead of using a single representation only which is suitable for all types. As a consequence, the compilation rules for the code generation must be parameterized with respect to types, i. e., the code generated for a concrete language construct must be adapted to the actual array types involved. Moreover, a shape-invariant argument may be applied to a shape-specific operation (or the other way round). Hence, it may be necessary to convert arguments from one representation into another. In order to get an near-optimal runtime performance, the different array representations should be designed in a way that minimizes the cost of these conversion operations.

To summarize, the work described in this thesis contributes the following to the state of the art:

- It introduces a compilation scheme for transforming arbitrary shape-invariant array operations into efficiently executable code. Shortcomings of recent compilers — which either restrict the expressive power of the language or generate code with unsatisfactory runtime performance only — are avoided.

- It extends this compilation scheme by general support for function overloading with respect to shapes. Whenever the overloading can not be resolved statically, additional code is generated which resolves the overloading at runtime.

- It provides an optimizing code generator. The code generator utilizes multiple array representations which have been individually adapted to the actual level of shape information. These measures lead to substantial runtime improvements compared to equivalent code which operates on a general array representation only.

- The compilation scheme described in this thesis has been fully integrated into the existing SAC compiler. In order to demonstrate its effectiveness, several runtime measurements have been performed on a Sun workstation and an Intel-based personal computer.

The remainder of this thesis is organized as follows: Chapter 2 gives a brief introduction to the programming language SAC. However, several parts of the language which are irrelevant for this thesis are omitted. Chapter 3 addresses the compilation scheme of the recent SAC compiler which is restricted to shape-specific code generation only. After a description of the major compilation steps, the chapter identifies the flaws of this compilation scheme and develops measures to eliminate them. The new compilation scheme which incorporates these extensions is introduced in Chapter 4. Subsequently, Chapter 5 evaluates the runtime performance of the code generated by the new compiler backend. Finally, Chapter 6 concludes the thesis and outlines some directions for future work.

Chapter 2

SAC — Single Assignment C

This chapter gives a brief introduction of the programming language SAC. Some parts of the language which are irrelevant for the remainder of this thesis are omitted, e. g., user-defined types, the module system [Grel96], and the class system for states and I/O [Grel96, GS95]. A more detailed description of SAC can be found in [Scho96, Scho03].

2.1 Functional Subset of C

The language kernel of SAC is a functional subset of the language C [KR88, Schi93]. The basic idea is to stick as close as possible to the syntax of C, but to restrict the set of legal programs in a way that allows a purely functional interpretation. The semantics of the language is then given by a rather straightforward mapping into an applied λ-calculus [Bare84, HS86].

This functional semantics rules out all elements of C that cause side-effects, e. g., global variables and pointers. As a consequence of dropping pointers, this kernel includes scalar data types only. Moreover, the control flow instructions `break`, `continue`, and `goto` have to be left out. For the remaining language constructs of C a sound functional interpretation can be found. For instance, assignments are considered nested `let`-bindings, and loops are internally transformed into tail-end recursive functions [Scho96].

The core syntax of SAC programs is shown in Figure 2.1. As in C, a program consists of a sequence of function definitions, among these a specific function `main` which serves as starting point for the program execution. The syntax of function definitions is adopted from C as well, but SAC functions may have multiple return values, thus, function headers may contain multiple return types,

```
Program      ⇒  [ FunDef ]* Main [ FunDef ]*
Main         ⇒  int main ( [ void ] ) FunBody
FunDef       ⇒  RetTypes Id ( [ Arguments ] ) FunBody
RetTypes     ⇒  void | Type [ , Type ]*
Arguments    ⇒  void | Type Id [ , Type Id ]*
FunBody      ⇒  { [ VarDec ]* [ Assign ]* Return }
Vardec       ⇒  Type Id ;
Assign       ⇒  Let ;
             |  if ( Expr ) AssignBlock [ else AssignBlock ]
             |  do AssignBlock while ( Expr )
             |  while ( Expr ) AssignBlock
             |  for ( Let ; Expr ; Let ) AssignBlock
AssignBlock  ⇒  Assign | { [ Assign ]* }
Let          ⇒  FunAp
             |  Id [ , Id ]* = Expr
             |  Id LetBinOp Expr
             |  Id LetUniOp | LetUniOp Id
FunAp        ⇒  Id ( [ Expr [ , Expr ]* ] )
             |  Expr BinOp Expr
             |  UniOp Expr
Expr         ⇒  FunAp | Id | Const | ( Expr )
LetBinOp     ⇒  += | -= | *= | /= | %=
LetUniOp     ⇒  ++ | --
BinOp        ⇒  + | - | * | / | %
             |  && | || | < | > | <= | >= | == | !=
UniOp        ⇒  + | - | !
Return       ⇒  return ( [ Expr [ , Expr ]* ] ) ;
Type         ⇒  int | float | double | bool | char
Id           ⇒  identifier
Const        ⇒  constant value
```

Figure 2.1: Core syntax of SAC programs (in BNF).

and `return` statements may contain multiple expressions. SAC explicitly distinguishes between integer and boolean values, hence, an additional base type `bool` is added. Furthermore, the type declarations for local variables in function bodies are optional — missing declarations are inferred by the compiler itself. Note here, that SAC supports *function overloading* [CW85], i. e., functions may share the same name as long as they differ with respect to the types or the number of their formal arguments.

2.2 Arrays in SAC

The SAC language kernel as described in the preceding section supports scalar data types only. It is now extended by a high-level array concept which is completely compatible with the functional paradigm, and which allows for an array oriented, less error-prone programming style which significantly improves program readability and code re-use.

2.2.1 Representation of Arrays

SAC supports the notion of multi-dimensional arrays as they are known from array languages such as APL [Iver62,ISO84,ISO93], J [Burk96], or NIAL [JJ93]. All arrays are represented by two vectors: a *data vector* containing all array elements in row-major order, and a *shape vector* which specifies the number of elements per axis. For reasons of uniformity scalars are considered arrays with empty shape. Figure 2.2 illustrates the array representation by means of four examples.

Data and shape vector cannot be entirely freely chosen. Let A be a n-dimensional array with shape vector $sv = [sv_0, \ldots, sv_{n-1}]$ and data vector $dv = [dv_0, \ldots, dv_{l-1}]$. Then, the length l of the data vector must be

$$l = \prod_{j=0}^{n-1} sv_j \quad .$$

Subarrays or elements of the array A may be addressed by *index vectors* of the set

$$\{\ [iv_0, \ldots, iv_{m-1}] \mid (0 \leq m \leq n) \wedge (\forall_{j \in \{0,\ldots,m-1\}} : 0 \leq iv_j < sv_j)\ \} \quad .$$

$A_0 = 7$

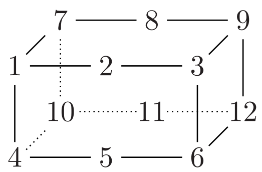
Dimension: 0
Shape vector: [ ]
Data vector: [7]

$$A_1 = \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix} = [7, 8, 9]$$

Dimension: 1
Shape vector: [3]
Data vector: [7, 8, 9]

$$A_2 = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

Dimension: 2
Shape vector: [2, 3]
Data vector: [7, 8, 9, 10, 11, 12]

$A_3 =$ 

Dimension: 3
Shape vector: [2, 2, 3]
Data vector: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Figure 2.2: Representation of arrays.

An index vector $iv = [iv_0, \ldots, iv_{m-1}]$ selects the subarray with shape vector $[sv_m, \ldots, sv_{n-1}]$ and data vector $[dv_p, \ldots, dv_{p+q-1}]$, where p and q satisfy

$$p = \sum_{j=0}^{m-1} \left(iv_j \cdot \prod_{k=j+1}^{n-1} sv_k \right) \quad , \quad q = \prod_{j=m}^{n-1} sv_j \quad .$$

The special cases $(m = 0)$ and $(m = n)$ specify the selection of the whole array A and the selection of the single array element dv_p respectively.

With these definitions at hand, a n-dimensional array in SAC may be defined as an expression of the form

$$\texttt{reshape([}sv_0\texttt{, ..., }sv_{n-1}\texttt{], [}dv_0\texttt{, ..., }dv_{l-1}\texttt{])} \qquad ,$$

where `reshape` is a built-in primitive (see next subsection), all sv_j are expressions evaluating to integer scalars, and all dv_j are expressions evaluating to scalars of identical type. For scalars and vectors of scalars the specification of the shape vector is optional:

$$\begin{array}{lcl} \texttt{reshape([], [}dv_0\texttt{])} & \equiv & dv_0 \\ \texttt{reshape([}l\texttt{], [}dv_0\texttt{, ..., }dv_{l-1}\texttt{])} & \equiv & \texttt{[}dv_0\texttt{, ..., }dv_{l-1}\texttt{]} \end{array} \qquad .$$

$$\begin{array}{lcl} \mathit{Expr} & \Rightarrow & \cdots \mid \mathit{VectExpr} \\ \mathit{VectExpr} & \Rightarrow & \texttt{[}\ [\ \mathit{Expr}\ [\ \texttt{,}\ \mathit{Expr}\ ]^*\ ]\ \texttt{]} \end{array}$$

Figure 2.3: Syntax of vector expressions (in BNF).

For convenience, the `[...]` notation of vectors may be used to specify higher-dimensional arrays as well. In fact, such vectors are expressions, and any legitimate SAC expression may be used to specify each of their elements (see Figure 2.3). However, all elements of a vector must evaluate to arrays of identical shape and type. For instance, an array of shape $[2, 3]$ may be considered as a vector of shape $[2]$ whose elements are vectors of shape $[3]$:

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

Let A and B denote arrays of shape $[3]$. Then, tupling and concatenation of A and B may be specified as follows:

`reshape( [2,3], [`A`,`B`])` $\equiv$ `[`A`,`B`]`
`reshape( [6], [`A`,`B`])` .

Note here, that arrays may be empty and that empty arrays can have manifold shapes in which at least one component is 0:

`reshape( [0], [])` $\equiv$ `[]`
`reshape( [1,0], [])` $\equiv$ `[ [] ]`
`reshape( [3,0,2], [])` .

2.2.2 Primitive Array Operations

For the purpose of this subsection, let A and val denote arbitrary expressions, and let iv and sv denote expressions that evaluate to vectors of proper length. SAC provides a small set of built-in array operations which are defined as follows:

- `dim(` A`)` returns the dimension of A.
- `shape(` A`)` returns the shape vector of A.

- `sel(` iv `,` A`)` returns a new array containing the subarray of A selected by iv. If iv does not represent a legitimate index vector with respect to A, the result of the operation is undefined.
- `reshape(` sv `,` A`)` returns a new array whose shape and data vector is identical to sv and the data vector of A respectively. If the product of the shape vector elements does not equal the length of the data vector, the result of the operation is undefined.
- `genarray(` sv `,` val`)` returns a new array whose shape vector is identical to the concatenation of sv and `shape(` val`)` . Its data vector is composed of repeated copies of the data vector of val.
- `modarray(` A `,` iv `,` val`)` returns a new array which is identical to A except for the subarray selected by iv which is set to val. If iv does not represent a legitimate index vector with respect to A, or the shape of the subarray does not equal the shape of val, the result of the operation is undefined.

All these operations are generically defined, i. e., they can be applied to arrays of arbitrary shape.

For the primitive operations `sel` and `modarray` exist alternative notations which are familiar from the language C. The expression `sel(` iv `,` A`)` may be written as A`[`iv`]` , and the assignment A`=modarray(` A `,` iv `,` val`);` may be replaced by A`[`iv`]=`val`;` .

2.2.3 With-Loop Construct

More complex array operations than the built-ins introduced in the preceding subsection may be defined in SAC by means of the `with`-loop. This language construct typically defines an entire array along with a specification of how to compute each array element depending on its index position. In this regard, the `with`-loop is similar to array comprehensions in other functional languages, like HASKELL or CLEAN, and to the `for`-loop in SISAL. However, `with`-loops in SAC allow the specification of generic, i. e., truly shape-invariant, array operations.

The syntax of `with`-loops is outlined in Figure 2.4. A `with`-loop basically consists of two parts: a *generator* part and an *operation* part. The generator part defines a set of index vectors along with an index variable representing elements of this set. Two expressions that must evaluate to vectors of equal lengths define the lower and upper bound of a range of index vectors. This set of

Expr	⇒	... | *WithExpr*
WithExpr	⇒	`with` `(` *Generator* `)` [`{` [*Assign*]* `}`] *Operation*
Generator	⇒	*Bound* *RelOp* *Id* *RelOp* *Bound* [*Filter*]
Bound	⇒	`.` | *Expr*
RelOp	⇒	`<=` | `<`
Filter	⇒	`step` *Expr* [`width` *Expr*]
Operation	⇒	`genarray` `(` *Expr* `,` *Expr* `)`
	|	`modarray` `(` *Expr* `,` *Id* `,` *Expr* `)`
	|	`fold` `(` *Id* `,` *Expr* `,` *Expr* `)`

Figure 2.4: Syntax of `with`-loops (in BNF).

index vectors may be restricted by an optional filter to define grids of arbitrary strides and widths. More precisely, let a, b, s, and w denote expressions that evaluate to vectors of length n, and let a_j, b_j, ... denote the j-th components of these vectors. Then, the generator

$$\texttt{(}a \texttt{ <= } iv \texttt{ < } b \texttt{ step } s \texttt{ width } w\texttt{)}$$

defines an index variable iv that can be referenced within the `with`-loop and whose domain is the following set of index vectors:

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

The operation part specifies the operation to be performed for each element of the index vector set. There are three different operations whose functionalities are defined as follows. Let sv denote an expression that evaluates to a vector, and let A and $expr$ denote arbitrary expressions. Moreover, let $foldop$ be the name of a binary commutative and associative function with neutral element $neutral$. Then,

- `genarray(` sv `,` $expr$ `)` generates a new array whose shape vector is identical to the concatenation of sv and `shape(` $expr$ `)` . Its elements are the values of $expr$ for all index vectors from the specified set, and arrays of shape `shape(` $expr$ `)` filled with zeros otherwise.
- `modarray(` A `,` iv `,` $expr$ `)` generates a new array of the same shape as A whose elements are the values of $expr$ for all index vectors from the specified set, and the values of A`[`iv`]` otherwise.

- `fold(` *foldop* `,` *neutral* `,` *expr* `)` specifies a reduction operation. Starting off with *neutral*, the value of *expr* is computed for each index vector from the specified set and subsequently folded using *foldop*. Associativity and commutativity of *foldop* guarantee determinate results irrespective of a particular evaluation order.

To increase program readability, an optional block of local assignments may be added between generation and operation part. This allows for the abstraction of complex subexpressions from the operation part. Moreover, in generators of `genarray`- and `modarray`-`with`-loops a dot (`.`) may replace one or both bound expressions. Depending on its syntactical position, it either represents the lowest or the highest legal index vector with respect to the shape of the result.

The expressive power of the `with`-loop allows to implement all the primitive array operations known from other languages like APL or FORTRAN-90 as library functions in SAC itself. The SAC standard array library provides, among others, functions for concatenation of arrays, shifting/rotating operations, and data reduction like sum, product, minimum or maximum. Moreover, all scalar operations are extended to arguments of arbitrary shapes, e. g., `([1,2,3] - 1)` evaluates to `[0,1,2]` .

2.2.4 Type System: Hierarchy of Array Types

For each base type (`int`, `float`, `double`, `bool`, `char`) SAC provides an entire hierarchy of array types. The most specific array types specify an exact shape whereas more general types prescribe an exact dimension only or contain no shape information at all.

The syntax of array types in SAC is given in Figure 2.5. Basically, an array type consists of a base type followed by a shape vector. If the shape is not completely defined, the components of the shape vector may be replaced by wildcards **.**, `*`, and `+`. The wildcard **.** means that the extent of a certain axis is unknown. The wildcards `+` and `*` represent arrays with at least dimension 1 or completely unknown dimension respectively.

This unbounded hierarchy of array types can be classified into four major categories:

- **Scalar arrays**, i. e., $\alpha \equiv \alpha[\,]$, where α denotes a base type. The set of all these types is represented by $\mathcal{T}_{\mathrm{SCL}}$ (SCL ≡ <u>sc</u>a<u>l</u>ar).

Type	⇒	*BaseType* `[*]`
	|	*BaseType* `[+]`
	|	*BaseType* `[ . [ , . ]* ]`
	|	*BaseType* `[` *Num* `[ ,` *Num* `]* ]`
	|	*BaseType* `[ ]`
	|	*BaseType*
BaseType	⇒	`int` | `float` | `double`
	|	`bool` | `char`

Figure 2.5: Syntax of array types (in BNF).

- **Non-scalar arrays with known dimension and known extent**, e. g., $\alpha[1]$, $\alpha[2,3]$. The set of all these types is represented by $\mathcal{T}_{\text{AKS}}$ (AKS $\equiv$ array of known shape).

- **Non-scalar arrays with known dimension but unknown extent**, e. g., $\alpha[\bullet]$, $\alpha[\bullet,\bullet]$. The set of all these types is represented by $\mathcal{T}_{\text{AKD}}$ (AKD $\equiv$ array of known dimension).

- **Arrays with unknown dimension and unknown extent**, e. g., $\alpha[+]$, $\alpha[*]$. The set of all these types is represented by $\mathcal{T}_{\text{AUD}}$ (AUD $\equiv$ array of unknown dimension).

The induced subtype relation — which is reflexive, transitive, and antisymmetrical — is depicted in Figure 2.6 as a directed tree (Hasse diagram). Vertices of the tree represent types and an edge leading from $\mathcal{T}$ to σ means that σ is a subtype of $\mathcal{T}$. The dashed lines separate the four categories of array types.

At first glance, the types $\alpha[*]$ and $\alpha[+]$ for arrays with unknown dimension seem to be rather artificial. Most user-defined functions tend to operate on arrays of fixed dimension only, e. g., the function `Det` which is defined in the following section. Nevertheless, being able to define truly dimension-invariant functions is a very important feature of SAC. It allows to restrict the set of built-in operations to a minimum, and to implement more complex primitive array operations in SAC itself. The advantage of this approach is twofold. First, it leads to a lean compiler layout since the compiler has to support only a small number of built-in operations. Moreover, it gives the user the opportunity to define new primitives or even to modify the existing ones.

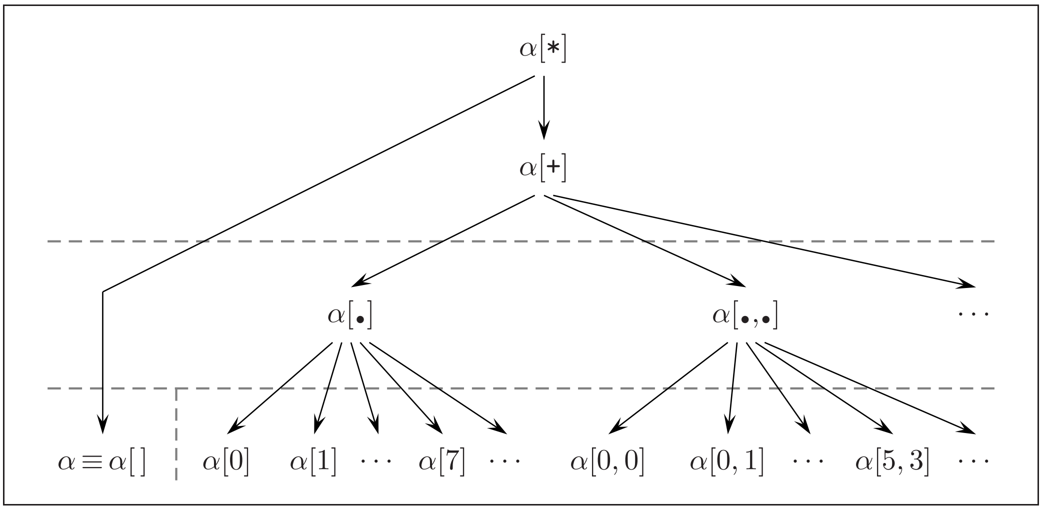


Figure 2.6: Hierarchy of array types.
(Hasse diagram of the subtype relation.)

2.3 Example: Determinant

Consider as an example computing the determinant of a two-dimensional array. For arrays with shape $[2, 2]$ this operation is very simple:

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc \quad . \tag{2.1}$$

Higher-order determinants may be computed recursively using the Laplace expansion (along the first column) [BSMM99]:

$$\det(A) = \sum_{i=0}^{n-1} (-1)^i \cdot A_{i0} \cdot \det\left(\mathfrak{A}_{i0}\right) \quad , \tag{2.2}$$

where A is an array of shape $[n, n]$, A_{ij} denotes the element of A at index position $[i, j]$, and $\mathfrak{A}_{ij}$ represents the array A without the i-th row and j-th column.

This mathematical specification of the algorithm could be translated into a single recursive SAC function as illustrated in Figure 2.7, where the two assignment blocks of the conditional (lines 4 and 7) implement Equation (2.1) and Equation (2.2) respectively.

Now, suppose that the programmer wants to add a non-recursive implementation for arrays of shape $[3, 3]$, or that he wants to include support for

```
int Det( int[.,.] A)
{
  if( shape( A) == [2,2]) {
    ret = ...;
  }
  else {
    ret = ... Det( ...) ...;
  }

  return( ret);
}
```

Figure 2.7: Computing the determinant of a two-dimensional array without function overloading.

argument shapes $[0, 0]$ and $[1, 1]$. In both situations he may modify the function `Det` accordingly. But this programming style has a serious conceptual flaw, since each extension of the function requires its source code to be available. This shortcoming could be avoided by using *function overloading* with respect to shapes.

Figure 2.8 depicts an alternative implementation of the algorithm using two instances of the function `Det`, one for each of the Equations (2.1) and (2.2). The first instance (lines 1 – 4) is suitable for arrays of shape $\texttt{int}[2, 2]$ only, the second one (lines 6 – 15) applies to all two-dimensional integer arrays with bigger shapes. Here, the explicit conditional of the first implementation is replaced by the implicit overloading mechanism, which makes the code more concise. Moreover, it is possible to add additional instances of `Det` for other argument shapes without recompiling the existing ones.

The Laplace expansion is implemented using a `fold-with`-loop (lines 9 – 12) whose generator defines the index vector set $\{[0, 0] \leq \texttt{iv} \leq [n-1, 0]\}$. For each of these index vectors $\texttt{iv} = [i, j]$ first the array $\mathfrak{A}_{ij}$ is generated by means of another function `Elim`, and subsequently, the product $(-1)^i \cdot A_{ij} \cdot \det(\mathfrak{A}_{ij})$ is calculated. The operation part of the `with`-loop computes the sum of all these values.

Note here, that the function `Elim` is defined in a very generic way in order to facilitate the reuse in a different context. The first argument A is not necessarily a two-dimensional array but may have arbitrary shape. The sec-

```
int Det( int[2,2] A)
{
  return( A[[0,0]] * A[[1,1]] - A[[1,0]] * A[[0,1]]);
}

int Det( int[.,.] A)
{
  sv = shape( A) - 1;
  ret = with ([0,0] <= iv <= [sv[[0]],0]) {
          B = Elim( A, iv);
          val = pow( -1, iv[[0]]) * A[iv] * Det( B);
        } fold( +, 0, val);

  return( ret);
}

int[*] Elim( int[*] A, int[.] pos)
{
  sv = shape( A) - 1;
  ret = with (. <= iv <= .) {
          new_iv = where( (iv < pos), iv, (iv + 1));
          val = A[ new_iv];
        } genarray( sv, val);

  return( ret);
}

int main()
{
  A = ...;
  ret = Det( A);

  return( ret);
}
```

Figure 2.8: Computing the determinant of a two-dimensional array with function overloading.

ond argument pos should evaluate to a vector with length `dim(` A `)`. Then, `Elim(` A `,` pos `)` generates a new array containing the elements of A where in each axis i all elements with index pos`[`i`]` have been omitted. The implementation of `Elim` uses the standard library function `where` which is defined as follows. Let $cond$, a, and b denote expressions that evaluate to vectors of equal length n, where $cond$ has the base type `bool`. That being the case, `where(` $cond$ `,` a `,` b `)` creates a vector of length n, and the i-th element of the data vector is set to a`[`i`]` if $cond$`[`i`]` evaluates to `true` and set to b`[`i`]` otherwise.

This example of a SAC program is well-suited to illustrate the basic problems that arise when trying to compile generic SAC code into efficiently executable code. Therefore, it will be used as a running example for the remainder of this thesis.

Chapter 3

Compilation of SAC Programs into Non-Generic Code

This chapter describes the compilation scheme of the recent SAC compiler (revision v0.9.1). Section 3.1 briefly introduces the major steps in compiling generic, shape-invariant SAC programs into non-generic and efficiently executable code. Subsequent sections discuss the most important compilation phases in more depth and demonstrate their effect by means of the SAC program introduced in the preceding chapter (Figure 2.8). Finally, Section 3.6 identifies the flaws of this compilation scheme and develops measures to eliminate them.

3.1 Outline of the Compilation Process

The major phases of the recent SAC compiler are shown in Figure 3.1. After *scanning and parsing* a SAC program, some *code simplifications* are performed. For instance, nested expressions are eliminated by adding temporary variables, assignment operators (for example `a += 2`) are replaced by their regular counterparts (`a = a + 2`), and `for`-, `do`-, and `while`-loops are transformed into tail-end recursive functions. These measures reduce the variety of language constructs and, hence, simplify subsequent compilation steps. Unfortunately, they also reduce the readability of intermediate SAC code, therefore, the effects of these simplifications will be ignored for the code examples given in the following sections.

The next compilation phase infers the types of all local variables. In order to achieve best possible potential for code optimizations, the compiler specializes all array types to specific shapes. As a consequence, generically defined

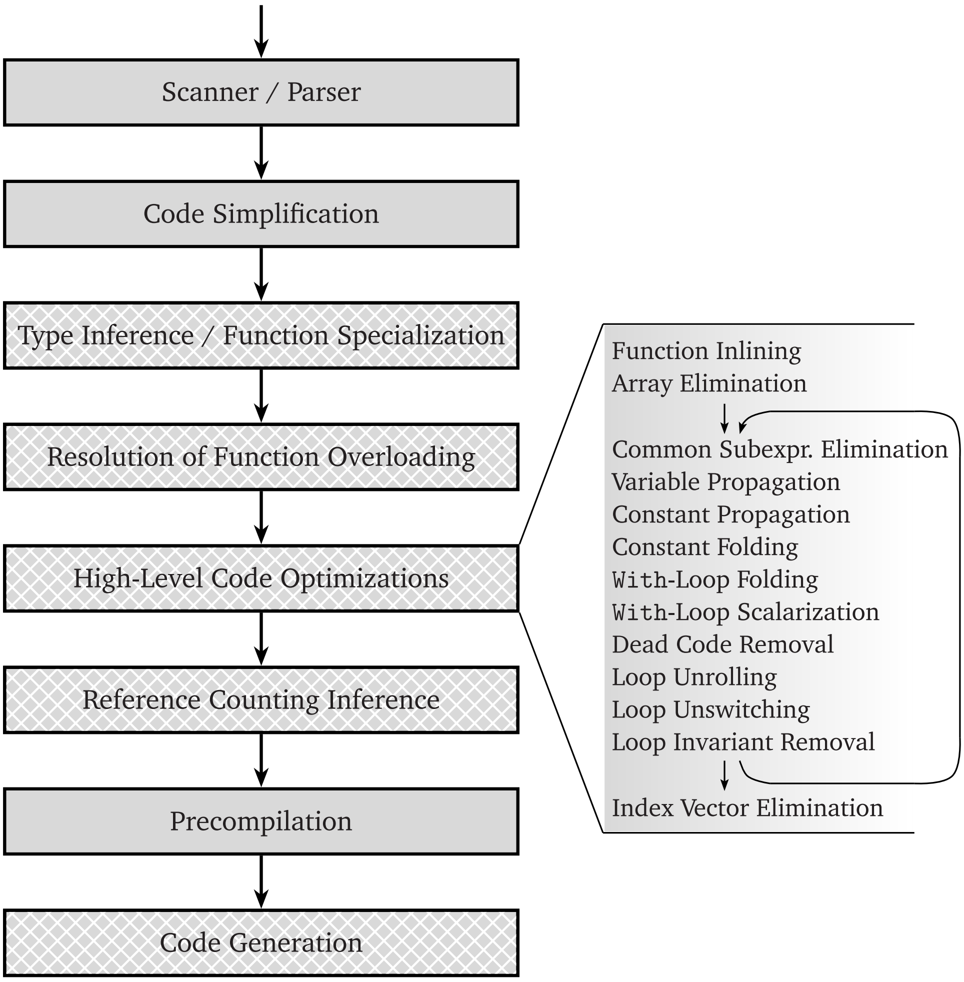


Figure 3.1: Outline of the compilation process.

functions have to be specialized for all required argument shapes. Whenever this *static shape inference* fails, an error message is issued and the compilation process is aborted — in such cases the programmer must add proper type declarations or function specializations by hand. A more detailed explanation of this compiler phase will be given in Section 3.2.

Since the compiler proceeds only if all array shapes have been inferred, the next task — *resolution of function overloading* — is trivial. Knowing the argument shapes of an application, it is statically decidable which instance of the function has to be used.

Subsequently, the compiler applies several *high-level code optimizations*, which are briefly described in Section 3.3. Many of these optimizations interact with each other, e. g., having applied a particular optimization may enable another one, therefore, the optimizations are performed repeatedly, as depicted on the right hand side of Figure 3.1. This cycle terminates if either the code does not change anymore or a predefined number of cycles has been performed.

SAC provides an implicit memory management based on *reference counting* [Cohe81, Cann89, FO95], which identifies and removes garbage as soon as the last access to it has been made. For this purpose, the compiler adds operations to the SAC code that handle the reference counters at runtime (see Section 3.4).

The *precompilation* phase is inverse to the loop-transformations performed during the code simplification phase. It converts tail-end recursive functions back into loops, which is crucial for the runtime performance of the compiled code.

Finally, the *code generation* is done. In order to liberate the compiler from all hardware-specific particularities, C is used as target language for the compilation. Because of the strong syntactical and semantical similarity between SAC and C, the code generation is almost trivial for many language constructs. The most important task of this phase is to find adequate array representations and to generate optimized code for array operations. These issues will be addressed in Section 3.5.

3.2 Type Inference and Function Specialization

Basically, the inference algorithm works as follows: Starting from the designated function `main`, the type inference system traverses all function bodies from outermost to innermost, propagating shapes as far as possible. Whenever a function application is encountered, it has to be determined which function definition is *relevant* for it, i. e., which instance of a possibly overloaded function must be used to compute the result of the application. Then, the type of the application equals the return type of this function instance. Furthermore, if the relevant function instance is a generic one, it is specialized with respect to the inferred argument shapes.

Take as an example the implementation of the function `Det` given in Figure 2.8 on page 20. Let the variable `A` in line 30 denote an array of type `int`$[3,3]$. In that case, for the application of `Det` in line 31 the second instance is relevant and the type inference system generates a specialized `int`$[3,3]$ version of this instance. Hence, the argument `A` in line 6 has shape $[3,3]$ and the application in line 10 enforces the specialization of the function `Elim`. Subsequently, the compiler deduces that the variable `B` represents an array of shape $[2,2]$, as a consequence, for the application of `Det` in line 11 the first instance is relevant. The result of these transformations is depicted in Figure 3.2, with all modified code fragments printed in a different color. Note that the compiler has added proper declarations for the local variables and that all instances of the function `Det` have unique names now. The latter is done by adding suffixes representing the types of the arguments (e. g., `__i_3_3` for `int`$[3,3]$).

The inference algorithm as outlined above has some important implications which will be formalized in the following. To do so, some notations have to be defined: $(\mathcal{T} \preceq \sigma)$ denotes that $\mathcal{T}$ is a subtype of σ, $(\mathcal{T} \prec \sigma)$ indicates that $\mathcal{T}$ is a proper subtype of σ, i. e., it is an abbreviation for $(\mathcal{T} \preceq \sigma) \wedge (\mathcal{T} \neq \sigma)$. The formula $(\mathcal{T} \not\prec \sigma)$ stands for the negation of $(\mathcal{T} \prec \sigma)$, and $x : \mathcal{T}$ means that for the variable x the type $\mathcal{T}$ has been inferred.

Now, consider an application of an overloaded function f with N instances to arguments $x_1, \ldots, x_m$ which have already been passed through the type inference system and have produced types $\mathcal{T}_1, \ldots, \mathcal{T}_m$:

$$f\,(\; x_1 : \mathcal{T}_1 \,,\; \ldots,\; x_m : \mathcal{T}_m) \quad .$$

In order to infer the type of this application, it has to be determined which instance of f is relevant for the application. Let

$$\sigma_1^k, \ldots, \sigma_n^k\; f^{(k)}(\; \mathcal{T}_1^k\; a_1, \ldots, \mathcal{T}_m^k\; a_m)\; \{\; Body\; \} \quad , \quad k \in \{1, \ldots, N\}$$

denote all the instances of f that occur in the given SAC program, where $\mathcal{T}_i^k$ denotes the i-th argument type and σ_j^k denotes the j-th return type of the k-th instance. A function instance $f^{(k)}$ is said to be *relevant* for the above application

```
int Det__i_2_2( int[2,2] A)
{ ... }

int Det__i_3_3( int[3,3] A)
{
  int[2] iv;
  int[2,2] B;
  int val;
  int ret;

  ...
  ret = with ( ... iv ... ) {
          B = Elim__i_3_3( A, iv);
          val = ... * Det__i_2_2( B);
        } fold( +, 0, val);

  return( ret);
}

int[2,2] Elim__i_3_3( int[3,3] A, int[2] pos)
{ ... }

int main()
{
  int[3,3] A;
  int ret;

  A = ...;
  ret = Det__i_3_3( A);

  return( ret);
}
```

Figure 3.2: SAC *program after type inference and function specialization.*

if and only if two conditions hold:

$$\forall_{i\in\{1,\ldots,m\}} : \mathcal{T}_i \preceq \mathcal{T}_i^k \quad \text{, and}$$

$$\neg \left(\exists_{l\in\{1,\ldots,N\}\setminus\{k\}} : \forall_{i\in\{1,\ldots,m\}} : \mathcal{T}_i \preceq \mathcal{T}_i^l \preceq \mathcal{T}_i^k \right) \quad .$$

The first condition ensures that actual and formal arguments of the application have compatible types, i. e., one is a subtype of the other. The second condition excludes all instances which are redundant, since it is guaranteed that always another instance l with more specific argument shapes can be found.

Without loss of generality, let

$$\left\{ f^{(k)} \mid k \in \{1, \ldots, R\} \right\} \quad , \quad R \leq N$$

denote the set of relevant function instances. As a prerequisite for resolving the function overloading uniquely, the cardinality R of this set should be 0 or 1. In the former case the compilation process is aborted with an error message complaining about a missing function definition, in the latter case the type inference is continued with a specialized version of $f^{(1)}$. Unfortunately, R may indeed be greater 1. Consider as an example an overloaded function `fun` with the following two instances given by the programmer:

```
int fun( int[.] A, int[2] B) { ... }
int fun( int[2] A, int[.] B) { ... }
```

.

With regard to the argument `A`, the second instance has a more specific type than the first one, with regard to the argument `B`, it is the other way round. Consider an application of `fun` where for both arguments the type `int`$[2]$ has been inferred. Then, both of the two function instances are relevant for this application and without additional criteria it is undecidable which one has to be chosen, hence, the function overloading is ambiguous here. This problem could be solved by introducing different priorities for different argument positions, but then, the semantics of SAC programs would depend on the order of the function arguments. To avoid confusion about the overloading mechanism, instances k and l with such argument types — compatible types in all argument positions i, more specific type in a first argument position i_1, and less specific type in a second argument position i_2 — are ruled out:

$$\neg \Big(\exists_{k,l\in\{1,\ldots,N\}} : \left(\forall_{i\in\{1,\ldots,m\}} : \mathcal{T}_i^k \preceq \mathcal{T}_i^l \ \vee \ \mathcal{T}_i^k \succ \mathcal{T}_i^l \right) \wedge \\ \left(\exists_{i_1\in\{1,\ldots,m\}} : \mathcal{T}_{i_1}^k \succ \mathcal{T}_{i_1}^l \right) \wedge \left(\exists_{i_2\in\{1,\ldots,m\}} : \mathcal{T}_{i_2}^k \prec \mathcal{T}_{i_2}^l \right) \Big) \quad ,$$

For the same reason, no pairwise distinct function instances should have identical argument signatures:

$$\neg \left(\exists_{k,l \in \{1,\ldots,N\}; k \neq l} : \forall_{i \in \{1,\ldots,m\}} : \tau_i^k = \tau_i^l \right) \quad .$$

Under these conditions, it is guaranteed that the function overloading can be resolved uniquely, and the type of the j-th return value of the given application is equal to the corresponding return type σ_j^1 of the relevant function instance.

3.3 High-Level Code Optimization

The SAC compiler performs several high-level code optimizations, most of which are well-known in compiler design [ASU86, BGS94, Appe98, AK02, Cann93]. Of particular interest, however, are four SAC-specific optimizations which try to get rid of arrays whenever they can either be entirely avoided or be replaced by scalars:

- ***With-loop folding*** eliminates intermediate arrays by transforming consecutive `with`-loops into single ones. This optimization technique is applicable even to `with`-loops which specify non-identical index vector sets in their generator parts. Such folding results cannot be expressed by a single `with`-loop, therefore, the SAC compiler internally uses a generalized version of `with`-loops that allows an arbitrary number of generator/operation pairs to be specified. More information about `with`-loop folding can be found in [Scho98b].

- ***With-loop scalarization*** eliminates intermediate arrays by transforming nested `with`-loops into non-nested ones. This optimization is particularly useful in the context of *axis control* [GS03] — a mechanism to focus the effects of array operations on (non-scalar) subarrays of higher-dimensional arrays, e. g., computing the element-wise, row-wise, or column-wise sum of a two-dimensional array.

- ***Array elimination*** replaces small arrays by sets of scalars, which in turn can be compiled into much more efficiently executable C code.

- ***Index vector elimination*** tries to replace index vectors by scalar offsets into data vectors. For example, let A denote an array of shape $[3, 3]$. Then, the expression A`[[2,1]]` selects the element at position

$(2 \cdot 3 + 1)$ of the data vector. Therefore, this expression may be replaced by A`{2*3+1}` , where the curly brackets `{·}` signify that a scalar offset rather than an index vector is used for the selection.

Note here, that the success of these optimizations critically depends on the static shape inference. The more specific the inferred array shapes the better is the effect of these optimizations.

The impact of this compiler phase on the example program is demonstrated in Figure 3.3. It turns out that the functions `Elim__i_3_3` and `Det__i_2_2` have been removed after inlining them. Moreover, the remaining function `Det__i_3_3` merely consists of simple arithmetical operations on scalars — all complex array operations, loops, and index vectors have been eliminated. In fact, this function represents a direct implementation of the Sarrus' rule for computing $3{\times}3$-determinants [BSMM99].

3.4 Reference Counting Inference

In SAC, arrays are treated as values, e. g., the same array may be referenced several times without loss of referential transparency. With regard to code generation, the most important objective is to avoid superfluous creation and copying of arrays, in order to generate code with a competitive runtime efficiency. This difficulty can be addressed by means of the so-called *reference counting* [Cohe81, Cann89, FO95] technique.

From a conceptual point of view multiple references of an array signify the existence of several copies. The basic idea of reference counting is to use virtual copies (pointers) rather than explicit copies of such an array. Only if one of these virtual copies needs to be modified, a real copy must be created. To keep track of the number of active references at runtime, for each array a reference counter is provided. Upon creation of an array it is initialized with the number of references in the current code block. Whenever a function is applied to an array, the corresponding reference counter is incremented by the number of references in the function body and subsequently decremented by 1. The increment represents the virtual copies of the array substituted in the function body, and the decrement represents the virtual copy which has been consumed by the function application.

The compiler has to insert these reference count operations during code generation. For this purpose, the compilation scheme makes use of an abstract function $\mathrm{Refs}(A)$ which returns for each variable A occurring on the left hand

```
int Det__i_3_3( int[3,3] A)
{
  ...               /* variable declarations */

  A00 = A{0*3+0};   /* A[[0,0]] */
  A01 = A{0*3+1};   /* A[[0,1]] */
  ...
  A22 = A{2*3+2};   /* A[[2,2]] */
  val1 =  1 * A20 * (A01 * A12 - A11 * A02);
  val2 = -1 * A10 * (A01 * A22 - A21 * A02);
  val3 =  1 * A00 * (A11 * A22 - A21 * A12);
  ret = val1 + val2 + val3;

  return( ret);
}

int main()
{
  int[3,3] A;
  int ret;

  A = ...;
  ret = Det__i_3_3( A);

  return( ret);
}
```

Figure 3.3: SAC *program after high-level code optimizations.*

side of an assignment or in the argument list of a function definition the number of references to it. This information is inferred in a separate compilation phase called *reference counting inference*.

A naive implementation of reference counting would increment and decrement the counters on every occurrence of a program variable. To avoid the ensuing runtime overhead wherever possible, the SAC compiler performs several reference count optimizations, which have been invented in the context of SISAL [SS88, Cann89]. However, as these optimizations are irrelevant in the context of this thesis, they will not be looked at.

3.5 Code Generation

In the final compilation phase the optimized and reference counted SAC code is transformed into ANSI C code. An important issue in this context is to find an appropriate C representation for SAC arrays, i. e., to translate the variable declarations of SAC programs into proper C declarations.

3.5.1 Array Representation

As a result of ruling out dynamically shaped arrays during compilation, the compiler backend needs to support arrays of known shapes only. In order to avoid unnecessary overhead, arrays that are identified as scalars are represented in C by scalar values as well. Other SAC arrays are uniquely defined by means of a shape vector and a data vector. In practice, the shape vector will hardly contain more than a few elements, and can therefore be implemented in C by a set of constant scalars rather than a vector.[1] Furthermore, a *reference counter* (short: rc) for the implicit memory management is needed. For convenience, the array representation provides an additional constant representing the size of the data vector.

Figure 3.4 illustrates the array representation by means of two examples. A scalar SAC variable A is simply translated into a scalar C variable of the same name. But if A denotes a two-dimensional array, six C variables have to be declared — the constants A`_dim`, A`_size`, A`_sv0`, and A`_sv1` contain the shape information, the variables A`_rc` and A`_data` represent the reference counter and the data vector respectively.

[1] Arrays with small data vectors have already been eliminated during the high-level code optimizations (array elimination).

Decl. in SAC	Declaration in C
$\alpha[]\ A$;	$\alpha\ A$;

$\alpha[4,3]\ A$;	const int A_dim = 2;
	const int A_size = 12;
	const int A_sv0 = 4;
	const int A_sv1 = 3;
	int *A_rc;
	α *A_data;

Figure 3.4: C *representations for scalar and non-scalar* SAC *arrays.*

3.5.2 Compilation Rules

With an appropriate array representation at hand, the code generation can be specified by means of a compilation scheme

$$\mathcal{C}\,[\![\ \text{SAC program}\ ]\!] \longmapsto \text{C program} \quad ,$$

which transforms SAC code into semantically equivalent C code. In order to liberate the compilation scheme from a concrete implementation, pseudo statements — so-called *intermediate code macros* (ICMs for short) — are used within the C code. On the one hand this allows to change the code generation without modifying the compiler itself, on the other hand it makes the compilation scheme more concise.

The following ICMs are repeatedly used by the compilation rules:

- `ALLOC(` $A : \mathcal{T}$ `)` allocates memory needed for the C representation of the array A, i. e., it allocates memory for the reference counter (A`_rc`) and the data vector (A`_data`) if $\mathcal{T}$ is a non-scalar type, and does nothing otherwise. Subsequently, this macro initializes the reference counter (if present) with 1.

- `FREE(` $A : \mathcal{T}$ `)` releases the array A (reference counter and data vector) from memory.

- `ADJUST_RC(` $A : \mathcal{T}$ `,` cnt`)` increments the reference counter (if present) of the array A by the integer cnt:

  $$*A\texttt{\_rc += } cnt; \quad .$$

Note that cnt could be a negative number which leads to a decrement of $(-cnt)$ of course. If the adjusted reference counter drops to 0, the array is no longer needed and therefore removed from memory:

```
if (*A_rc < 1) {
  FREE( A:T);
}
```
.

- `SV(` $A:\mathcal{T}$`)` returns the shape vector of the array A.
- `DV(` $A:\mathcal{T}$`)` returns the data vector of the array A.
- `SET_DV_SUB(` $A:\mathcal{T}$ `,` iv `,` dv`)` fills the part of `DV(` A`)` which is addressed by the index vector iv with the elements of the vector dv, where dv must be made to fit exactly into the addressed part of the data vector.
- `SET_DV(` $A:\mathcal{T}$ `,` dv`)` fills the data vector of the array A with the elements of the vector dv, i. e., it is an abbreviation for the statement `SET_DV_SUB(` $A:\mathcal{T}$ `,` $[\,]$, dv`)` .
- `SET_DV_PRF(` $B:\sigma$ `,` prf `,` $A_1:\mathcal{T}_1$ `, ... ,` $A_m:\mathcal{T}_m$`)` fills the data vector of the array B with the result of the primitive array operation prf applied to the arguments $A_1, \ldots, A_m$.

Furthermore, the operation $+\!\!+$ is used to concatenate two vectors.

The compilation rules for the basic language constructs are defined as follows. (Note, that for reasons of clarity, function definitions and applications are restricted to a single argument and a single return value here. Nevertheless, the SAC compiler can handle arbitrary numbers of arguments and return values of course.) The first rule applies to a sequence of function definitions:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} \sigma\ \ fun\texttt{(}\ \mathcal{T}\ \ A\texttt{)} \\ \texttt{\{} \\ \quad Vardecs \\ \quad Body \\ \quad \texttt{return(}\ B:\sigma\texttt{);} \\ \texttt{\}} \\ Rest \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \texttt{FUN\_DEC(}\ fun\texttt{,} \\ \qquad\qquad \texttt{DEC\_OUT(}\ B:\sigma\texttt{),} \\ \qquad\qquad \texttt{DEC\_IN(}\ A:\mathcal{T}\texttt{))} \\ \texttt{\{} \\ \quad \mathcal{C}[\![\ Vardecs\ ]\!] \\ \quad \texttt{DECL\_SHAPE\_ARG(}\ A:\mathcal{T}\texttt{)} \\ \quad \texttt{ADJUST\_RC(}\ A:\mathcal{T}\texttt{,}\ \mathrm{Refs}(A){-}1\texttt{)} \\ \quad \mathcal{C}[\![\ Body\ ]\!] \\ \quad \texttt{FUN\_RET(}\ B:\sigma\texttt{)} \\ \texttt{\}} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. .$$

The macro FUN_DEC defines the header of a C function fun with the return type void and formal arguments which are generated by the ICMs

$$\texttt{DEC\_OUT(}\ B:\mathcal{O}\texttt{)}\quad,\quad\texttt{DEC\_IN(}\ A:\mathcal{T}\texttt{)}\qquad,$$

i. e., the return values of a SAC function are implemented as reference parameters because C functions allow a single return value only. Suppose $\mathcal{T}$ and $\mathcal{O}$ are both non-scalar types with base types α and β respectively. Then, the macro DEC_IN stands for the two arguments

$$\texttt{int *}A\texttt{\_rc}\quad,\quad\alpha\ \texttt{*}A\texttt{\_data}\qquad,$$

whereas DEC_OUT stands for the two reference parameters

$$\texttt{int **ret\_}B\texttt{\_rc}\quad,\quad\beta\ \texttt{**ret\_}B\texttt{\_data}\qquad.$$

Consistent with these macros, the return statement of a SAC function is compiled into an ICM named FUN_RET which assigns these reference parameters:

$$\begin{array}{l}\texttt{*ret\_}B\texttt{\_rc}\quad\ \texttt{=}\ B\texttt{\_rc;}\\ \texttt{*ret\_}B\texttt{\_data = }B\texttt{\_data;}\end{array}\qquad.$$

Note that it is unnecessary to pass the shape information of the argument (A_dim, etc.) to the function. Instead, these values are declared as local constants by means of the macro DECL_SHAPE_ARG. Consider, for example, that $\mathcal{T}$ represents the type float$[4, 3]$. In this case, DECL_SHAPE_ARG would expand to the following C declarations:

$$\begin{array}{l}\texttt{const int }A\texttt{\_dim\ \ = 2;}\\ \texttt{const int }A\texttt{\_size = 12;}\\ \texttt{const int }A\texttt{\_sv0\ \ = 4;}\\ \texttt{const int }A\texttt{\_sv1\ \ = 3;}\end{array}\qquad.$$

The macro ADJUST_RC is used to update the reference counter of the argument A. As already mentioned in Section 3.4, the reference counter must be incremented by the number of references in the function body and subsequently decremented by 1, i. e., an increment of $(\mathrm{Refs}(A) - 1)$ is needed, where Refs denotes the abstract function which has been inferred during the reference counting inference phase of the compiler.

The next rule covers variable declarations:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\mathcal{T}\ \ A;\\ \mathit{Rest}\end{array}\right]\!\!\right]\longmapsto\left\{\begin{array}{l}\texttt{DECL(}\ A:\mathcal{T}\texttt{)}\\ \mathcal{C}\,[\![\ \mathit{Rest}\ ]\!]\end{array}\right.\qquad.$$

The macro DECL expands to a C declaration as shown in Figure 3.4 on page 33. Since the shape vectors of arrays are defined in the declaration part as constants, other compilation rules which generate new arrays have to deal with data vectors only.

Assignments with a variable on the right hand side

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B:\sigma\ \texttt{=}\ A:\tau\texttt{;} \\ Rest \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \texttt{ASSIGN(}\ B:\sigma\texttt{,}\ A:\tau\texttt{)} \\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right.$$

compile into the macro `ASSIGN` which simply does the following assignments:

$$\begin{array}{l} B\texttt{\_rc}\ \ \ \texttt{=}\ A\texttt{\_rc;} \\ B\texttt{\_data}\ \texttt{=}\ A\texttt{\_data;} \end{array} \qquad .$$

As a consequence, B and A represent the same array and the reference counter must be incremented by $(\mathrm{Refs}(B) - 1)$.

Assignments with a vector construct on the right hand side require the creation of a new array which is done by means of four ICM statements:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B:\sigma\ \texttt{= [}\ A_1:\tau\texttt{,}\ \ldots\texttt{,}\ A_m:\tau\ \texttt{];} \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{l} \texttt{ALLOC(}\ B:\sigma\texttt{)} \\ \texttt{SET\_DV(}\ B:\sigma\texttt{,}\ \texttt{DV(}\ A_1:\tau\texttt{)} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \texttt{DV(}\ A_m:\tau\texttt{))} \\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(}\ A_i:\tau\texttt{,}\ {-1}\texttt{)} \qquad , \qquad i \in \{1,\ldots,m\} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. \qquad .$$

`ALLOC` allocates memory for the reference counter as well as for the data vector of the new array and initializes the reference counter with 1. The macro `SET_DV` is used to write the entries of the data vector. Finally, the reference counters of the vector elements involved have to be adjusted. The counter of the new array is incremented by $(\mathrm{Refs}(B) - 1)$, and the counters of the arguments are decremented by 1.

Assignments with a constant scalar on the right hand side are compiled similar to the preceding rule:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B:\sigma\ \texttt{=}\ val\texttt{;} \\ Rest \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \texttt{ALLOC(}\ B:\sigma\texttt{)} \\ \texttt{SET\_DV(}\ B:\sigma\texttt{,}\ [val]\texttt{)} \\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. \quad .$$

Since B is a scalar array here, the two macros `ALLOC` and `ADJUST_RC` expand to empty statements, and `SET_DV` simply generates the assignment

$$B\ \texttt{=}\ val\texttt{;} \qquad .$$

Assignments with a function application are compiled into the macro `FUN_AP`, which is the counterpart of `FUN_DEC`:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\sigma \texttt{ = } fun\texttt{(} A\!:\!\mathcal{T}\texttt{);} \\ Rest \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \texttt{FUN\_AP(} fun\texttt{,} \\ \qquad\quad \texttt{AP\_OUT(} B\!:\!\sigma\texttt{),} \\ \qquad\quad \texttt{AP\_IN(} A\!:\!\mathcal{T}\texttt{))} \\ \texttt{ADJUST\_RC(} B\!:\!\sigma\texttt{, } \mathrm{Refs}(B){-}1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. .$$

The macro FUN_AP stands for an application of the function fun to arguments which are generated by the ICMs

$$\texttt{AP\_OUT(} B\!:\!\sigma\texttt{)} \quad , \quad \texttt{AP\_IN(} A\!:\!\mathcal{T}\texttt{)} \qquad .$$

Assuming that $\mathcal{T}$ and σ are both non-scalar types, the macro AP_IN expands to the two arguments

$$A\texttt{\_rc} \quad , \quad A\texttt{\_data} \qquad ,$$

and AP_OUT expands to the two reference arguments

$$\texttt{\&}B\texttt{\_rc} \quad , \quad \texttt{\&}B\texttt{\_data} \qquad .$$

Moreover, the reference counter of the result B has to be incremented by $(\mathrm{Refs}(B) - 1)$. The reference counters of the arguments, however, are left untouched — they are handled by the function itself.

The primitive operation reshape generates an array with the same data vector as the given argument. Since the shape vector of the result has already been defined in the declaration part (DECL), the reshape operation can in fact be compiled like an assignment with a variable on the right hand side:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\sigma \texttt{ = reshape(} sv\!:\!\mathcal{T}'\texttt{, } A\!:\!\mathcal{T}\texttt{);} \\ Rest \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \mathcal{C}[\![\ B\!:\!\sigma \texttt{ = } A\!:\!\mathcal{T}\texttt{;}\ ]\!] \\ \texttt{ADJUST\_RC(} sv\!:\!\mathcal{T}'\texttt{, } -1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. .$$

Besides, the reference counter of the (ignored) argument sv has to be decremented.

The primitive operation modarray returns a new array which is identical to the first argument except for a subarray. In general, it is necessary to allocate new memory for the result (ALLOC) and to initialize *all* elements of the new data vector (SET_DV_PRF). If the reference counter of the first argument equals 1, however, the modarray operation can be done in place, i. e., the first argument can be reused for the result (ASSIGN) and only the specified subarray of the data vector must be rewritten (SET_DV_SUB):

$$
\mathcal{C}\left[\!\left[\begin{array}{l} B:\sigma\ \texttt{ = modarray(}\ A:\tau\texttt{,}\ iv:\tau'\texttt{,}\ val:\tau''\texttt{);} \\ Rest \end{array}\right]\!\right]
\longmapsto \left\{\begin{array}{l}
\texttt{if(}\ A\texttt{\_rc} > 1\texttt{) \{} \\
\quad \texttt{ALLOC(}\ B:\sigma\texttt{)} \\
\quad \texttt{SET\_DV\_PRF(}\ B:\sigma\texttt{, modarray, }\ A:\tau\texttt{,}\ iv:\tau'\texttt{,}\ val:\tau''\texttt{)} \\
\quad \texttt{ADJUST\_RC(}\ A:\tau\texttt{,}\ -1\texttt{)} \\
\texttt{\}} \\
\texttt{else \{} \\
\quad \texttt{ASSIGN(}\ B:\sigma\texttt{,}\ A:\tau\texttt{)} \\
\quad \texttt{SET\_DV\_SUB(}\ B:\sigma\texttt{,}\ iv:\tau'\texttt{, DV(}\ val:\tau''\texttt{))} \\
\texttt{\}} \\
\texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\
\texttt{ADJUST\_RC(}\ iv:\tau'\texttt{,}\ -1\texttt{)} \\
\texttt{ADJUST\_RC(}\ val:\tau''\texttt{,}\ -1\texttt{)} \\
\mathcal{C}[\![\ Rest\ ]\!]
\end{array}\right. .
$$

Note that the compiler does not generate the `if`-clause if it is redundant. Whenever $Rest$ contains at least one reference of A, it is impossible to do the `modarray` operation in place, i. e., the condition $(A\texttt{\_rc} > 1)$ would always be satisfied. In this case, it suffices to generate the first code block of the `if`-clause only.

The remaining primitive array operations are compiled in a uniform way:

$$
\mathcal{C}\left[\!\left[\begin{array}{l} B:\sigma\ \texttt{ = }\ prf\texttt{(}\ A_1:\tau_1\texttt{,}\ \ldots\texttt{,}\ A_m:\tau_m\texttt{);} \\ Rest \end{array}\right]\!\right]
\longmapsto \left\{\begin{array}{l}
\texttt{ALLOC(}\ B:\sigma\texttt{)} \\
\texttt{SET\_DV\_PRF(}\ B:\sigma\texttt{,}\ prf\texttt{,}\ A_1:\tau_1\texttt{,}\ \ldots\texttt{,}\ A_m:\tau_m\texttt{)} \\
\texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\
\texttt{ADJUST\_RC(}\ A_i:\tau_i\texttt{,}\ -1\texttt{)} \quad , \quad i \in \{1,\ldots,m\} \\
\mathcal{C}[\![\ Rest\ ]\!]
\end{array}\right. .
$$

They all produce a new array, therefore, an `ALLOC` statement is needed. Subsequently, the data vector of this array is filled with the result of the array operation (`SET_DV_PRF`). After that, the reference counters have to be adjusted as usual.

The complete compilation scheme for `with`-loops is rather complicated and beyond the scope of this thesis. During code optimizations (`with`-loop folding) the compiler may create `with`-loops with multiple generator/operation pairs. Besides, the compiler backend allows to partition or alter the iteration space of `with`-loops, e. g., for *loop tiling* [Wolf89, LRW91] to improve cache performance, or for load balancing on multiprocessor systems. All these aspects are of no importance for the remainder of this thesis and will be ignored here. A detailed coverage of these issues can be found in [Krey98, GKS00] (sequentially executable code only) and [Grel01] (implicit support for multiprocessor systems).

Ignoring a few details, the following C code is generated for a `modarray`-`with`-loop:

$$
\mathcal{C}\left[\!\!\left[\begin{array}{l}
B:\sigma\ \texttt{= with(}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{) \{}\\
\qquad\qquad expr:\mathcal{T}''\ \texttt{=}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{;}\\
\qquad\quad\texttt{\} modarray(}\ A:\mathcal{T}\texttt{,}\ iv:\mathcal{T}'\texttt{,}\ expr:\mathcal{T}''\texttt{);}\\
Rest
\end{array}\right]\!\!\right]
\longmapsto
\left\{\begin{array}{l}
\texttt{ALLOC(}\ B:\sigma\texttt{)}\\
\texttt{WITH\_LOOP(}\ iv\texttt{,}\\
\quad\mathcal{C}[\![\ expr:\mathcal{T}''\ \texttt{=}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{;}\ ]\!]\\
\quad\texttt{SET\_DV\_SUB(}\ B:\sigma\texttt{,}\ iv\texttt{, DV(}expr:\mathcal{T}''\texttt{))}\\
\quad\texttt{ADJUST\_RC(}\ expr:\mathcal{T}''\texttt{,}\ -1\texttt{)}\\
\texttt{,}\\
\quad\texttt{COPY\_DV\_SUB(}\ B:\sigma\texttt{,}\ iv\texttt{,}\ A:\mathcal{T}\texttt{)}\\
\texttt{)}\\
\texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)}\\
\texttt{ADJUST\_RC(}\ A:\mathcal{T}\texttt{,}\ -1\texttt{)}\\
\mathcal{C}[\![\ Rest\ ]\!]
\end{array}\right. .
$$

The macros `ALLOC` and `ADJUST_RC` are used to create a new[2] array B and to adjust the reference counters of the arrays involved. The macro `WITH_LOOP` generates all the code needed to iterate over the whole index vector domain of the array B. The first argument of this macro defines the name of the iteration variable iv. The second argument specifies the code to be executed in each iteration step if the current value of iv is covered by the generator: The operation specified for the `with`-loop is performed and the result ($expr$) is stored in the array B by means of the macro `SET_DV_SUB`. The third argument defines the code to be executed for the complementary index vectors: The corresponding array elements are copied from the source array A, using the macro `COPY_DV_SUB`.

The compilation rule for `genarray`-`with`-loops is almost identical to the one for `modarray`-`with`-loops:

[2] Similar to the rule for the primitive operation `modarray`, the compiler may add branch conditions to reuse already allocated arrays rather than allocating new memory. For reasons of clarity, this has been left out here.

$$
\mathcal{C}\left[\!\!\left[\begin{array}{l} B:\sigma\ \texttt{= with(}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{) \{} \\ \qquad\qquad expr:\mathcal{T}''\ \texttt{=}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{;} \\ \qquad\quad \texttt{\} genarray(}\ sv:\mathcal{T}\texttt{,}\ expr:\mathcal{T}''\texttt{);} \\ Rest \end{array}\right]\!\!\right]
\longmapsto \left\{\begin{array}{l} \texttt{ALLOC(}\ B:\sigma\texttt{)} \\ \texttt{WITH\_LOOP(}\ iv\texttt{,} \\ \quad \mathcal{C}[\![\ expr:\mathcal{T}''\ \texttt{=}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{;}\ ]\!] \\ \quad \texttt{SET\_DV\_SUB(}\ B:\sigma\texttt{,}\ iv\texttt{,}\ \texttt{DV(}expr:\mathcal{T}''\texttt{))} \\ \quad \texttt{ADJUST\_RC(}\ expr:\mathcal{T}''\texttt{,}\ -1\texttt{)} \\ \texttt{,} \\ \quad \texttt{SET\_DV\_SUB(}\ B:\sigma\texttt{,}\ iv\texttt{,}\ [0,\ldots,0]\texttt{)} \\ \texttt{)} \\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(}\ sv:\mathcal{T}\texttt{,}\ -1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. .
$$

The main difference is found in the third argument of the WITH_LOOP macro. Array elements not covered by the generator are initialized with the default value 0 now.

The compilation of fold-with-loops is done as follows:

$$
\mathcal{C}\left[\!\!\left[\begin{array}{l} B:\sigma\ \texttt{= with(}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{) \{} \\ \qquad\qquad expr:\mathcal{T}''\ \texttt{=}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{;} \\ \qquad\quad \texttt{\} fold(}\ foldop\texttt{,}\ neutral:\mathcal{T}\texttt{,}\ expr:\mathcal{T}''\texttt{);} \\ Rest \end{array}\right]\!\!\right]
\longmapsto \left\{\begin{array}{l} \mathcal{C}[\![\ B:\sigma\ \texttt{=}\ neutral:\mathcal{T}\texttt{;}\ ]\!] \\ \texttt{WITH\_LOOP(}\ iv\texttt{,} \\ \quad \mathcal{C}[\![\ expr:\mathcal{T}''\ \texttt{=}\ \ldots\ iv:\mathcal{T}'\ \ldots\texttt{;}\ ]\!] \\ \quad \mathcal{C}[\![\ B:\sigma\ \texttt{=}\ foldop\texttt{(}\ B:\sigma, expr:\mathcal{T}''\texttt{);}\ ]\!] \\ \texttt{)} \\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(}\ neutral:\mathcal{T}\texttt{,}\ -1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. .
$$

Before entering the iteration, the result B is initialized with the given neutral element. Subsequently, for each element of the index vector set the value $expr$ is computed and folded into B. Note that it is unnecessary to specify a third argument for the macro WITH_LOOP here because the iteration space contains the generator elements only.

3.6 Towards an Enhanced Compilation Scheme

The compilation scheme described so far has been implemented in the recent SAC compiler (revision v0.9.1). Several case studies have shown that array-intensive applications written in SAC execute as fast as their FORTRAN or SISAL equivalents [Scho98a, GS00]. Unfortunately, the compilation scheme has a serious shortcoming: It relies on static shape inference (i. e., it is absolutely necessary to infer the exact shapes of all arrays encountered at compile time), but static shape inference is undesirable or even impossible in certain situations:

- Some input data may have variable shapes, e. g., if read from a file. In this case, the program has to be compiled individually for each shape configuration.
- Generic functions must be recompiled for each argument shape — no separate compilation (e. g., of libraries/modules) is possible for them.
- Each application of a generic function induces function specialization. This could be very time and space consuming in case of recursive functions whose argument shapes are changing with each recursive call. For example, applying the function `Det` (see Figure 2.8 on page 20) to an argument of shape $[100, 100]$ causes the compiler to build specialized instances for shape $[100, 100]$, $[99, 99]$, $[98, 98]$, etc. — these are 98 new instances in total!
- Even worse, static shape inference is undecidable in general.

In summary, on the one hand static knowledge of exact shapes is essential for generating code with a competitive runtime performance, on the other hand insistence on static shape inference restricts the expressive power of the language significantly. In order to resolve this dilemma, a compilation scheme has to be invented that supports not only exact shapes but the entire hierarchy of array types. It should generate shape-specific code whenever possible and create more generic code otherwise.

Such an approach basically brings about two problems. The first problem is about function overloading. A SAC program might contain shape-specific as well as generic instances of a single operation. The semantics of SAC prescribes that for each function application the most specific instance suitable for the arguments must be used. The question is, how this dispatch can be done.

Consider, for instance, applications of the function `Det` to arguments of type `int`$[3, 3]$, `int`$[\bullet,\bullet]$, and `int`$[+]$. For the first application the compiler may build a specialization of the second instance with argument shape $[3, 3]$ and resolve the overloading statically. For the second application no specialization is needed, but it is not statically decidable which instance must be used. Each of the three instances available — the two given by the programmer and the one built by the compiler — may be applicable. Thus, the compiler should generate additional code that chooses the matching instance at runtime. For the third application the situation is basically the same, but here an additional dynamic type check is recommended to ensure at runtime that the argument represents indeed a two-dimensional array.

So, in general it can be a rather complicated and time consuming task to determine which functions have to be specialized for which argument shapes, and to build all the tailor-made code fragments that resolve the overloading.

A word about "overloading": In principle, the compiler has to distinguish between overloaded instances given by the programmer and specialized instances built during the compilation process. The former instances *must* be dispatched since they (most likely) have different meanings. The latter instances *should* be dispatched — for efficiency reasons it is desirable to use specialized instances whenever possible, but ignoring them would not alter the meaning of the application. Take as an example an application of the function `Det` to an argument of type `int[•,•]`. It is essential here to take the user-defined instances (`Det__i_2_2`, `Det__i_X_X`) into account; it would be fatal to apply `Det__i_X_X` to an argument of shape $[2, 2]$. But it would be possible to ignore the specialized instance (`Det__i_3_3`) for the dispatch; applying `Det__i_X_X` instead of `Det__i_3_3` to an argument of shape $[3, 3]$ would reduce the runtime performance but has no impact on the result of the computation. From a pragmatic point of view, however, the distinction between user-defined and specialized instances is superfluous. Since runtime performance plays an important role, specialized instances are always dispatched in the same way as user-defined instances. Therefore, in this thesis the term "function overloading" is used for all functions for which multiple instances exist — irrespective of whether these instances are user-defined or compiler-generated (via specialization).

The second problem comes about in the code generation phase of the compiler. The compiler must be able to generate code for four categories of array types with different levels of shape information. For this purpose, suitable C representations for these shape informations must be found. Runtime evaluations reveal that it is more efficient to use four different representations (one for each type category), instead of a single one only. As a consequence, the compilation rules for the code generation must be parameterized with respect to types,

i. e., the code generated for a concrete language construct must be adapted to the actual array types involved. Unfortunately, the number of code variants increases exponentially with the number of arguments, e. g., if a language construct requires three arguments with four different array representations each, the compiler must be able to generate $4^3 = 64$ different code variants for it.

Moreover, formal and actual argument type of a function application may differ. Hence, it may be necessary to convert arguments from one representation into another. Take as an example an application of `Det` to an argument of type `int`$[\bullet,\bullet]$ which at runtime turns out to be a `int`$[2, 2]$ array. In this case, the first instance of `Det` has to be applied. But before doing so, the argument must be converted from the `int`$[\bullet,\bullet]$ into the `int`$[2, 2]$ representation. In order to get an optimal runtime performance, the different array representations should be designed in a way that minimizes the cost of these conversion operations.

Chapter 4

Compilation of SAC Programs into Generic Code

This chapter describes a new compilation scheme for SAC that supports the entire hierarchy of array types and, thus, avoids the flaws (see Section 3.6) of a compiler that relies on static shape inference only. The basic idea is to generate shape-specific C code whenever exact shapes can be statically inferred and to emit more generic code otherwise.

The basic structure of the new compilation scheme can be adopted from Section 3.1, however, some compilation phases have to be modified, namely the *type inference system* (see Section 4.1), the *resolution of function overloading* (see Section 4.2), and the *code generation* (see Section 4.3). A brief description of the actual implementation of the new compilation scheme along with some directions for further improvements can be found in Section 4.4.

Note, that the following section about type inference and function specialization is just an excerpt from [Scho01] and is given here as background information only. Topic of this thesis is code generation; it is demonstrated how the partial shape information provided by the type system can be exploited to generate efficiently executable code even in case of failing static shape inference. Hence, the innovative parts of the compilation scheme are described in Section 4.2 and Section 4.3.

4.1 Type Inference and Function Specialization

The basic inference algorithm is more or less the same as described in Section 3.2. The most important difference relates to the handling of function applications. Whenever a function application is encountered, it has to be determined which function instances are relevant for it. If the shapes of the arguments have been successfully inferred, there exists at most one relevant instance, as shown in Section 3.2. In general, however, this is not the case. Consider as an example that the function `Det` (see Figure 2.8 on page 20) is applied to an argument of type `int`$[\bullet,\bullet]$. For this application both instances of `Det` would be relevant. If the argument turns out to be a `int`$[2,2]$ array at runtime, the first instance must be used, otherwise the second instance should be applied. In order to approximate the type of the application, the return types of all relevant instances must be taken into account. Since this approximation should be as shape-specific as possible, the most specific common supertype (mscs for short) of the return types in question is used. Besides, specialization is enforced for each of the relevant instances.[1]

More formally, the type inference of applications is performed as follows. Consider an application of a function f to arguments $x_1, \ldots, x_m$ which have already been passed through the type inference system and have produced types $\mathcal{T}_1, \ldots, \mathcal{T}_m$:

$$f\,(\;x_1 : \mathcal{T}_1\,,\;\ldots,\;x_m : \mathcal{T}_m) \quad .$$

Let again

$$\sigma_1^k,\ldots,\sigma_n^k\; f^{(k)}\,(\;\mathcal{T}_1^k\; a_1\,,\ldots,\mathcal{T}_m^k\; a_m)\;\{\;Body\;\} \quad , \quad k \in \{1,\ldots,N\}$$

denote all the instances of f that occur in the given SAC program. The k-th instance of f is said to be *relevant* for the above application if and only if:

$$\forall_{i\in\{1,\ldots,m\}} : \left(\mathcal{T}_i \preceq \mathcal{T}_i^k \;\vee\; \mathcal{T}_i \succ \mathcal{T}_i^k\right) \quad \text{, and}$$
$$\neg\left(\exists_{l\in\{1,\ldots,N\}\setminus\{k\}} : \forall_{i\in\{1,\ldots,m\}} : \left(\mathcal{T}_i \preceq \mathcal{T}_i^l \preceq \mathcal{T}_i^k \;\vee\; \mathcal{T}_i \succ \mathcal{T}_i^l = \mathcal{T}_i^k\right)\right) \quad .$$

This definition of the term *relevant* is a generalized version of the definition given on page 28 — the difference being an additional disjunction term in each of the two formulas. Since the type inference system infers not only shape-specific but arbitrary types now, the actual argument type $\mathcal{T}_i$ may be a proper supertype of the formal argument type $\mathcal{T}_i^k$, e. g., applying `Det` to an argument

[1] In order to prevent code explosion, the compiler may decide to refrain from specialization in certain situations, e. g., if the number of instances exceeds a predefined bound, or if multiple relevant instances have been found and, hence, the specializations built may never be actually used at runtime.

of type `int`$[\bullet,\bullet]$ means that the instance with formal type `int`$[2,2]$ is relevant as well.

Let

$$\left\{f^{(k)} \mid k \in \{1,\ldots,R\}\right\} \quad , \quad R \leq N$$

denote the set of all relevant function instances. As already mentioned, the type of the j-th return value of the given application is

$$\text{mscs}\left(\left\{\sigma_j^k \mid k \in \{1,\ldots,R\}\right\}\right) \quad .$$

Obviously, such a supertype does not always exist, e. g., `int`$[+]$ and `float`$[\bullet]$ have no common supertype. In order to ensure that the type of the application exists, the type inference system has to check whether the return values of all relevant instances have pairwise a common supertype:

$$\forall_{k,l\in\{1,\ldots,R\}} : \forall_{j\in\{1,\ldots,n\}} : \exists\sigma : \left(\sigma \succeq \sigma_j^k \;\wedge\; \sigma \succeq \sigma_j^l\right) \quad .$$

Note here, that a common supertype of two arbitrary types exists if and only if the two types have an identical base type α — in worst cases the common supertype would be $\alpha[*]$. Thus, the preceding constraint can be simplified to:

$$\forall_{k,l\in\{1,\ldots,R\}} : \forall_{j\in\{1,\ldots,n\}} : \text{basetype}(\sigma_j^k) = \text{basetype}(\sigma_j^l) \quad .$$

4.2 Resolution of Function Overloading

The type inference system of the compiler infers for each function application the set of relevant function instances. If this set contains more than a single element, the compiler must generate additional code for the function application which dynamically chooses the matching instance at runtime. Moreover, it may not be statically inferable whether or not a function application is type-correct. Type-correctness is guaranteed if and only if all actual argument types are subtypes of the formal argument types. Otherwise the compiler has to generate additional code for dynamic type checks as well.

Fortunately, it turns out that it is not necessary to explicitly generate individual code that performs the resolution of overloading and the dynamic type checks for each function application. Instead, it suffices to generate it for the most general case only. This code is written in SAC itself and inserted into the SAC program via a high-level code transformation. Subsequently, individual and optimized code for each function application is obtained by means of the usual high-level code optimizations already integrated into the compiler, like function inlining, constant propagation, and constant folding.

4.2.1 Wrapper Functions: An Example

As an example consider a function `main` that contains applications of the function `Det` to arguments of type `int`$[3,3]$, `int`[•,•], and `int`[+]. Figure 4.1 depicts the SAC code with resolved function overloading and explicit type checks. There exist three instances of the function `Det`: Two of them have been given by the programmer (see lines 1, 2), the third one with argument shape $[3,3]$ (line 3) is built by the type inference system via specialization of the [•,•] version. All three instances of `Det` have unique names now.

The resolution code is implemented as a wrapper function `Det__i` with the most general argument type `int`[*] (lines 5 – 24). All applications of the function `Det` in the SAC source code are replaced by applications of this wrapper (lines 2, 3, 33 – 35). The wrapper selects the appropriate instance with respect to the actual shape of the argument (lines 10, 13, 16), or causes a runtime error if no appropriate instance has been found (line 20). In order to minimize on average the number of comparisons needed to choose the correct function instance, the choice is narrowed down by first checking the argument's dimension (line 8).

The interesting part of this code transformation is the generation of the wrapper functions which will be addressed in the next subsection.

The impact of the usual high-level code optimizations on the code fragment given above is demonstrated in Figure 4.2 — with all modifications compared to Figure 4.1 printed in a different color. It turns out that the three wrapper applications in the function `main` have been transformed in the expected way: The first application has been replaced by a direct call of the appropriate instance `Det__i_3_3` (line 16). The second application has been inlined and the redundant outermost `if`-clause of the wrapper code has been removed. Only the optimized code of the two inner `if`-clauses remain which tests whether the shape of the argument `B` is $[2,2]$ or $[3,3]$ (lines 18 – 30). The third application has been left as is since the unspecific shape of the argument `C` inhibits further optimizations (line 32).

4.2.2 Generation of Wrapper Functions

Consider a function f and let

$$\sigma_1^k, \ldots, \sigma_n^k \; f^{(k)}(\, \tau_1^k \; a_1, \ldots, \tau_m^k \; a_m) \; \{ \; Body \; \} \quad , \quad k \in \{1, \ldots, N\}$$

```
int Det__i_2_2( int[2,2] A) { ... }
int Det__i_X_X( int[.,.] A) { ... Det__i( B) ... }
int Det__i_3_3( int[3,3] A) { ... Det__i( B) ... }

/* wrapper function */
int Det__i( int[*] A)
{
  if (dim( A) == 2) {
    if (shape( A) == [2,2]) {
      ret = Det__i_2_2( A);
    }
    else if (shape( A) == [3,3]) {
      ret = Det__i_3_3( A);
    }
    else {
      ret = Det__i_X_X( A);
    }
  }
  else {
    ret = ERROR( "type error");
  }

  return( ret);
}

int main()
{
  int[3,3] A;
  int[.,.] B;
  int[+] C;
  ...

  a = Det__i( A);
  b = Det__i( B);
  c = Det__i( C);
  ...
}
```

Figure 4.1: SAC *program after resolution of function overloading.*

```
int Det__i_2_2( int[2,2] A) { ... }
int Det__i_X_X( int[.,.] A) { ... }
int Det__i_3_3( int[3,3] A) { ... }

/* wrapper function */
int Det__i( int[*] A)
{ ... }

int main()
{
  int[3,3] A;
  int[.,.] B;
  int[+] C;
  ...

  a = Det__i_3_3( A);

  /* b = Det__i( B); */
  svB = shape( B);
  svB0 = svB{0};
  svB1 = svB{1};
  if ((svB0 == 2) && (svB1 == 2)) {
    b = Det__i_2_2( B);
  }
  else if ((svB0 == 3) && (svB1 == 3)) {
    b = Det__i_3_3( B);
  }
  else {
    b = Det__i_X_X( B);
  }

  c = Det__i( C);

  ...
}
```

Figure 4.2: SAC program after high-level code optimizations.

denote all the instances of f that occur in the given SAC program. Generating the wrapper functions for f is a process that consists of three steps.

First, function overloading with respect to base types is resolved. In general, the instances of f may have different base type signatures, e. g., the operation + is defined on arguments of base type `int`, `float`, and `double`. Since the compiler has inferred the base type signatures of all function applications, this overloading can be resolved statically. Therefore, it suffices to create a separate wrapper function for each base type signature. As a consequence, it can be assumed here, without loss of generality, that all instances of f have identical base type signatures, i. e.,

$$\forall_{k,l \in \{1,\ldots,N\}} : \forall_{i \in \{1,\ldots,m\}} : \mathrm{basetype}(\mathcal{T}_i^k) = \mathrm{basetype}(\mathcal{T}_i^l) \quad .$$

In the second step, a decision tree is constructed which maps a given actual argument shape signature to the relevant instance of f, if it exists. Based on this decision tree a rather simple transformation scheme can be defined which generates the code for the wrapper function.

Before going into formal details, the function `Det` will be used to illustrate the basic structure of such a decision tree. The three instances of the function `Det` have formal argument types `int`$[2,2]$, `int`$[3,3]$, and `int`$[\bullet,\bullet]$. The decision tree reflects the subtype relation on these types together with the most general type `int`$[*]$, i. e., it is a directed tree consisting of four vertices. Moreover, each vertex $\mathcal{T}$ of the tree is labeled with a list containing those instances of `Det` which can be applied to all arguments of type $\mathcal{T}$ without causing a type error. Since the decision tree is meant to find the most specific instance applicable, this list is sorted with respect to subtyping. The rudimentary decision tree described so far is depicted in Figure 4.3, where $\mathcal{T}\!:\!\mathit{funs}$ represents a vertex $\mathcal{T}$ along with the label funs. The four labeled vertices of the tree relate to the following facts:

- `int`$[2,2]$: Both the instances `Det__i_2_2` and `Det__i_X_X` could be applied to arguments of shape $[2,2]$. But the instance `Det__i_2_2` is the more specific one.

- `int`$[3,3]$: Both the instances `Det__i_3_3` and `Det__i_X_X` could be applied to arguments of shape $[3,3]$. But the instance `Det__i_3_3` is the more specific one.

- `int`$[\bullet,\bullet]$: Two-dimensional arguments whose shape is neither $[2,2]$ nor $[3,3]$ could be applied to the instance `Det__i_X_X` only.

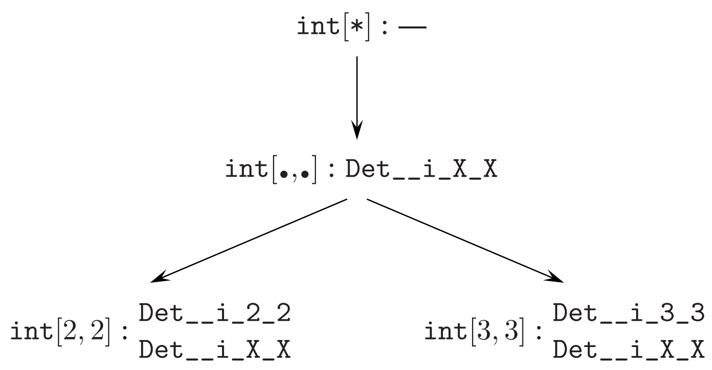


Figure 4.3: Rudimentary decision tree for the function `Det`*.*

- `int[*]`: Arguments whose dimension is different from 2 could not be applied to any instance of `Det`.

The function `Det` is a very simple example since it has a single argument only. Nevertheless, the same method can be applied to the general case — a function with m arguments — as well. Looking at each argument separately, it can be used to generate m partial decision trees. Subsequently, these trees must be combined properly to get the complete decision tree for all arguments.

More formally, the partial decision trees for the function f are defined as follows. For each base type α let

$$T_{[\alpha]} = \big(V_{[\alpha]}\,,\,E_{[\alpha]}\big)$$

denote the (infinite) directed tree of the subtype relation for α as depicted in Figure 2.6 on page 18, where the sets $V_{[\alpha]}$ and $E_{[\alpha]} \subset V_{[\alpha]}{\times}V_{[\alpha]}$ represent the vertices and edges of the tree respectively:

$$V_{[\alpha]} = \{\mathcal{T} \mid \mathrm{basetype}(\mathcal{T}) = \alpha\} \quad ,$$

$$E_{[\alpha]} = \big\{(\mathcal{T},\sigma) \in V_{[\alpha]}{\times}V_{[\alpha]} \mid (\mathcal{T}{\succ}\sigma) \wedge \neg\,(\exists_{\rho\in V_{[\alpha]}} : \mathcal{T}{\succ}\rho{\succ}\sigma)\big\} \quad .$$

This tree is called the Hasse diagram [BSMM99] of the relation; it forms the structural basis for partial decision trees. Roughly speaking, a partial decision tree is a finite part of a Hasse diagram with added vertex labels.

For each argument position i of the function f let α_i denote the base type of the formal argument types $\mathcal{T}_i^k$, and define a tree T_i^f which is constructed analogous to $T_{[\alpha_i]}$ but contains only the vertices $\mathcal{T}_i^k$ along with the root $\alpha_i[*]$:

$$T_i^f = \left(V_i^f \,,\, E_i^f \right) \quad , \quad i \in \{1, \ldots, m\} \quad ,$$

where

$$\begin{aligned} V_i^f &= \left\{ \mathcal{T}_i^k \mid k \in \{1, \ldots, N\} \right\} \cup \left\{ \alpha_i[*] \right\} \quad , \\ E_i^f &= \left\{ (\mathcal{T}, \sigma) \in V_i^f \times V_i^f \mid (\mathcal{T} \succ \sigma) \wedge \neg \left(\exists_{\rho \in V_i^f} : \mathcal{T} \succ \rho \succ \sigma \right) \right\} \quad . \end{aligned}$$

Note here, that the vertex $\alpha_i[*] \in V_i^f$ is crucial to guarantee the tree property of T_i^f. Since the number of function instances is finite by definition, this tree is — in contrast to $T_{[\alpha_i]}$ — finite as well. T_i^f represents the structure of the partial decision tree for argument position i. Now, the required vertex labels have to be added.

For that purpose, define a mapping $funs_i^f$ on the set of vertices V_i^f:

$$funs_i^f : \mathcal{T} \mapsto \left\{ \left\langle k \,,\, \Delta(\mathcal{T}_i^k, \mathcal{T}) \right\rangle \mid k \in \{1, \ldots, N\} \wedge \left(\mathcal{T}_i^k \succeq \mathcal{T} \right) \right\} \quad ,$$

where $\Delta(\mathcal{T}, \sigma)$ denotes the number of pairwise distinct vertices which lie on the path from $\mathcal{T}$ to σ (σ excluded), e. g.,

$$\begin{aligned} \Delta(\,\texttt{float}[+]\,,\, \texttt{float}[+]\,) &= 0 \quad , \\ \Delta(\,\texttt{float}[*]\,,\, \texttt{float}[\bullet]\,) &= 2 \quad . \end{aligned}$$

A vertex $\mathcal{T}$ labeled with a set containing the pair $\langle k, d \rangle \in funs_i^f(\mathcal{T})$ indicates that — with respect to the i-th argument position — the k-th instance of f can be applied to all arguments of type $\mathcal{T}$ without causing a type error, and that the number of proper subtypes which exist between the formal argument type $\mathcal{T}_i^k$ and the actual argument type $\mathcal{T}$ equals d. This number d — called the *distance* of an argument — will be used to identify the most specific instance suitable for a given argument shape signature.

For each vertex $\mathcal{T}$ of the tree T_i^f, the set $funs_i^f(\mathcal{T})$ could be considered a label for $\mathcal{T}$, hence,

$$\overline{T}_i^f = \left(V_i^f \,,\, E_i^f \,,\, funs_i^f \right)$$

denotes a (vertex-) labeled version of the tree T_i^f. The labeled tree $\overline{T}_i^f$ is the partial decision tree for argument position i of the function f.

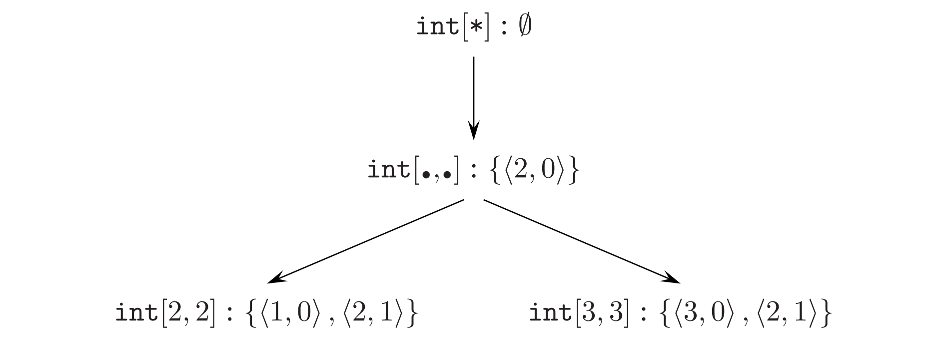


Figure 4.4: Labeled tree $\overline{T}_1^{\texttt{Det}}$.

Again, consider as an example the function `Det`. Let the indices 1, 2, and 3 represent the instances `Det__i_2_2`, `Det__i_X_X`, and `Det__i_3_3` respectively. `Det` has a single argument only, i. e., $m = 1$. Moreover, the set $V_1^{\texttt{Det}}$ contains the following types:

$$V_1^{\texttt{Det}} = \{\, \texttt{int}[*]\, ,\, \texttt{int}[\bullet,\bullet]\, ,\, \texttt{int}[2,2]\, ,\, \texttt{int}[3,3]\, \} \quad .$$

Figure 4.4 depicts the tree $\overline{T}_1^{\texttt{Det}}$, where $\mathcal{T}$: $\{\ldots\}$ represents a vertex $\mathcal{T}$ along with the label $\mathit{funs}_1^{\texttt{Det}}(\mathcal{T}) = \{\ldots\}$. Note that this tree is basically the same as the one given in Figure 4.3.

Since the function `Det` has a single argument only, $\overline{T}_1^{\texttt{Det}}$ is in fact already the decision tree needed to generate the wrapper code. In general, however, the separate trees $\overline{T}_1^f$, …, $\overline{T}_m^f$ for individual arguments must be combined properly. Take as an example the following two instances of a function `Foo`:

$$\begin{array}{l} \texttt{int Foo}^{(1)}\texttt{(int[.] A, int[.] B) \{ ... \}} \\ \texttt{int Foo}^{(2)}\texttt{(int[2] A, int[3] B) \{ ... \}} \end{array} \quad .$$

The corresponding trees $\overline{T}_1^{\texttt{Foo}}$ and $\overline{T}_2^{\texttt{Foo}}$ are shown in Figure 4.5. Assume that the wrapper tests the arguments in ascending order (i. e., $i = 1, 2$). Then, the tree $\overline{T}_1^{\texttt{Foo}}$ (left hand side of the figure) is a decision tree for the first argument. Guided by the type of the first argument, $\overline{T}_2^{\texttt{Foo}}$ (right hand side of the figure) can be used to build a decision tree for the second argument. If the first argument has the type `int`$[2]$, for example, the vertex label indicates that both instances of `Foo` are applicable. Therefore, $\overline{T}_2^{\texttt{Foo}}$ can be used as decision tree for the second argument. However, if the type of the first argument is `int`$[3]$, the label of the corresponding vertex `int`$[\bullet]$ tells that only the first instance of `Foo` is

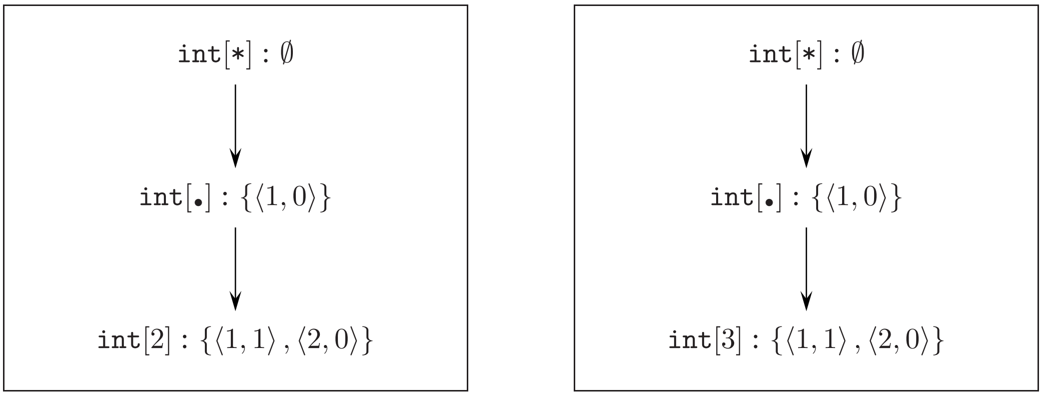


Figure 4.5: Labeled trees $\overline{T}_1^{\texttt{Foo}}$ *and* $\overline{T}_2^{\texttt{Foo}}$*.*

applicable. In this case, the instance $\texttt{Foo}^{(2)}$ must not be taken into account for the second argument anymore and should be removed from $\overline{T}_2^{\texttt{Foo}}$ to get a proper decision tree. These individual decision subtrees could be considered as additional vertex labels called $nextarg$.

The complete decision tree for the function `Foo` is depicted in Figure 4.6. Each vertex of the tree has a second label $nextarg$ (right hand sides of the arrows $\mapsto$) which specifies the decision tree for the next argument. Most of these trees are empty (./.) owing to the fact that no relevant instances are left or the last argument has been reached. The tree in the lower right corner is basically a copy of $\overline{T}_2^{\texttt{Foo}}$, but with an important modification: The labels $funs$ contain the *sum* of distances over both arguments now, rather than the distance of the second argument only. These distance sums will be used to choose the correct function instance for the wrapper code — the element of $funs$ with the most specific argument shapes will get the lowest distance sum. The tree in the upper right corner is a copy of $\overline{T}_2^{\texttt{Foo}}$ which has been adapted for the vertex $\texttt{int}[\bullet]$, i. e., the vertex $\texttt{int}[3]$ has been removed from the tree since it is relevant for $\texttt{Foo}^{(2)}$ only, all pairs with 2 as first component have been removed from the codomain of the mapping $funs$, and the labels $funs$ contain accumulated distances now.

In general, the decision tree for the function f has the form

$$\overline{T}^f = \left(V_1^f \,,\, E_1^f \,,\, funs_1^f \,,\, nextarg_1^f \right) \quad ,$$

i. e., it is the tree $\overline{T}_1^f$ together with new labels $nextarg_1^f$ which can be computed by means of an abstract algorithm $\mathcal{A}_{NextArg}$:

$$nextarg_1^f(\mathcal{T}) = \mathcal{A}_{NextArg} \left[\!\!\left[\; 1 \,,\, funs_1^f(\mathcal{T}) \; \right]\!\!\right] \quad , \quad \mathcal{T} \in V_1^f \quad .$$

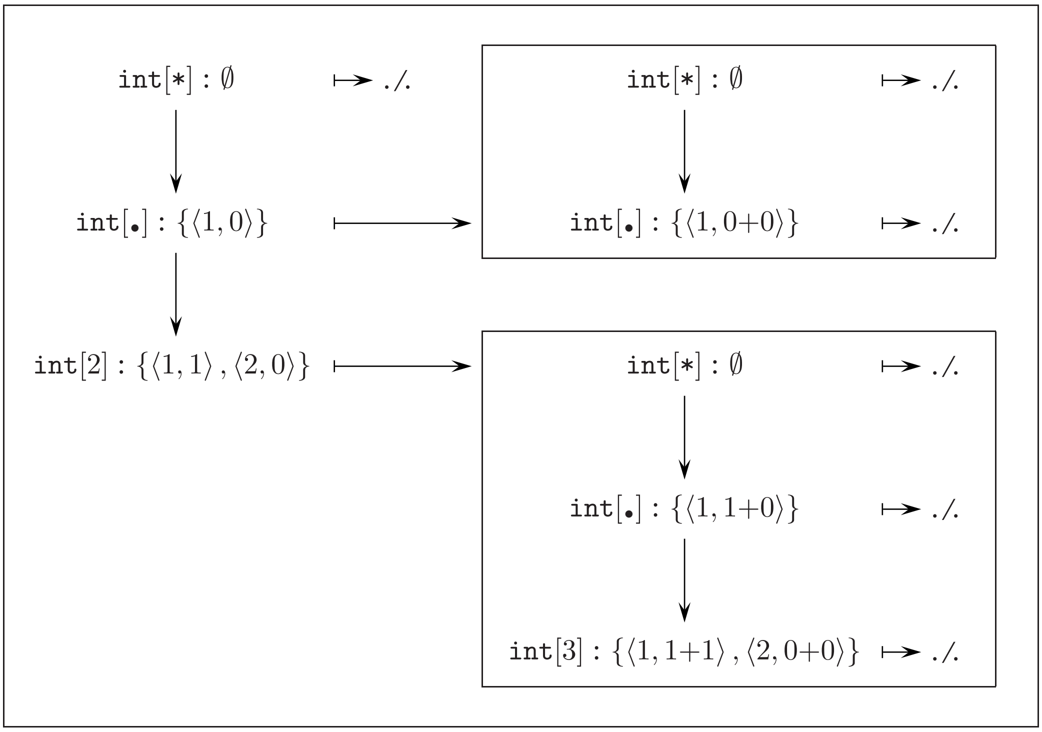


Figure 4.6: Decision tree for the function `Foo`.

This algorithm takes two arguments — an argument position i as well as a vertex label Funs — and is recursively defined as

$$\mathcal{A}_{\mathit{NextArg}} [\![\; i \; , \; \mathit{Funs} \;]\!] = \Big(V' \; , \; E' \; , \; \mathit{funs}' \; , \; \mathit{nextarg}' \Big) \quad ,$$

where

$$\begin{aligned} V' &= \big\{ \mathcal{T}^k_{i+1} \mid k \in \mathit{Funs} \big\} \cup \big\{ \alpha_{i+1}[*] \big\} \quad , \\ E' &= \big\{ (\mathcal{T},\sigma) \in V' \times V' \mid (\mathcal{T} \succ \sigma) \wedge \neg\, (\exists \rho{\in}V' : \mathcal{T} \succ \rho \succ \sigma) \big\} \quad , \\ \mathit{funs}' : \mathcal{T} &\mapsto \big\{ \langle k \; , \; d{+}d' \rangle \mid \big(\langle k, d \rangle \in \mathit{Funs} \big) \wedge \big(\langle k, d' \rangle \in \mathit{funs}^f_{i+1}(\mathcal{T}) \big) \big\} \quad , \\ \mathit{nextarg}' : \mathcal{T} &\mapsto \mathcal{A}_{\mathit{NextArg}} [\![\; i{+}1 \; , \; \mathit{funs}'(\mathcal{T}) \;]\!] \quad , \end{aligned}$$

i. e., (V', E', funs') is a labeled tree which is constructed similar to $\overline{T}^f_{i+1}$ but has been adapted for the label Funs. Having the definition of $\overline{T}^f_{i+1}$ in mind, the mapping funs' given above can be specified directly by

$$\mathit{funs}' : \mathcal{T} \mapsto \big\{ \big\langle k \; , \; d + \Delta(\mathcal{T}^k_{i+1}, \mathcal{T}) \big\rangle \mid \big(\langle k, d \rangle \in \mathit{Funs} \big) \wedge \big(\mathcal{T}^k_{i+1} \succeq \mathcal{T} \big) \big\} \quad .$$

The recursion ends if the label Funs represents an empty set or if the condition $(i = m)$ is hold. In the former case it would be useless to compute the labels $\mathit{nextarg}'$ since no applicable instance of f is left, and in the latter case the last argument position is reached. In both situations, the algorithm $\mathcal{A}_{\mathit{NextArg}}$ simply produces an empty tree:

$$\mathcal{A}_{\mathit{NextArg}} [\![\; i \; , \; \emptyset \;]\!] = \big(\, \emptyset \; , \; \emptyset \; , \; \perp \; , \; \perp \, \big) \quad ,$$

and

$$\mathcal{A}_{\mathit{NextArg}} [\![\; m \; , \; \mathit{Funs} \;]\!] = \big(\, \emptyset \; , \; \emptyset \; , \; \perp \; , \; \perp \, \big) \quad .$$

With a decision tree $\overline{T}^f$ at hand, a transformation scheme

$$\mathcal{C}_{\mathit{Wrap}} \Big[\!\!\Big[\; 1 \; , \; \overline{T}^f \; \Big]\!\!\Big] \longmapsto \text{SAC code}$$

is used to create the body of the wrapper function. The first parameter of this transformation scheme keeps track of the current argument position $i \in \{1, \ldots, m\}$ and the second parameter represents the decision tree which is traversed in postorder.

The rules for this transformation scheme are defined as follows. The first rule applies to decision trees with roots of the form $\alpha[*]$:

$$\mathcal{C}_{Wrap}\left[\!\!\left[\ i\ ,\ \begin{array}{ccc} & \alpha[*] : Funs, Nextarg & \\ \swarrow & \downarrow & \cdots \\ \overline{T}_1 & \overline{T}_2 & \end{array} \right]\!\!\right]$$

$$\longmapsto \left\{ \begin{array}{l} \mathcal{C}_{Wrap}[\![\ i\ ,\ \overline{T}_1\ ]\!] \\ \mathcal{C}_{Wrap}[\![\ i\ ,\ \overline{T}_2\ ]\!] \\ \cdots \\ \texttt{\{} \\ \quad \mathcal{C}_{NextArg}[\![\ i\ ,\ Funs\ ,\ Nextarg\ ]\!] \\ \texttt{\}} \end{array} \right. ,$$

where $Funs$ and $Nextarg$ denote the labels of the root and $\overline{T}_1, \overline{T}_2, \ldots$ represent (possibly missing) subtrees. At first, these subtrees are traversed and thereby transformed into a nested conditional. Subsequently, a second transformation scheme $\mathcal{C}_{NextArg}$, which will be defined later on, is used to generate the `else`-part of this conditional.

The three remaining rules have a rather similar structure — they all generate a nested conditional with unspecified `else`-part. The following rule is suitable for decision subtrees with roots of the form $\alpha[+]$:

$$\mathcal{C}_{Wrap}\left[\!\!\left[\ i\ ,\ \begin{array}{ccc} & \alpha[+] : Funs, Nextarg & \\ \swarrow & \downarrow & \cdots \\ \overline{T}_1 & \overline{T}_2 & \end{array} \right]\!\!\right]$$

$$\longmapsto \left\{ \begin{array}{l} \texttt{if(dim(}a_i\texttt{) > 0) \{} \\ \quad \mathcal{C}_{Wrap}[\![\ i\ ,\ \overline{T}_1\ ]\!] \\ \quad \mathcal{C}_{Wrap}[\![\ i\ ,\ \overline{T}_2\ ]\!] \\ \quad \cdots \\ \quad \texttt{\{} \\ \qquad \mathcal{C}_{NextArg}[\![\ i\ ,\ Funs\ ,\ Nextarg\ ]\!] \\ \quad \texttt{\}} \\ \texttt{\}} \\ \texttt{else} \end{array} \right. ,$$

The next rule applies to decision subtrees with roots of the form $\alpha[\bullet, \bullet, \ldots]$:

$$\mathcal{C}_{Wrap}\left[\!\!\left[\; i\,,\; \begin{array}{ccc} \multicolumn{3}{c}{\alpha[\overbrace{\bullet,\ldots,\bullet}^{d}] : \mathit{Funs}, \mathit{Nextarg}} \\ \swarrow & \downarrow & \ldots \\ \overline{T}_1 & \overline{T}_2 & \end{array} \;\right]\!\!\right]$$

$$\longmapsto \left\{ \begin{array}{l} \texttt{if(dim(}a_i\texttt{) == } d\texttt{) \{} \\ \quad \mathcal{C}_{Wrap}[\![\; i\,,\, \overline{T}_1 \;]\!] \\ \quad \mathcal{C}_{Wrap}[\![\; i\,,\, \overline{T}_2 \;]\!] \\ \quad \ldots \\ \quad \texttt{\{} \\ \qquad \mathcal{C}_{NextArg}[\![\; i\,,\; \mathit{Funs}\,,\; \mathit{Nextarg} \;]\!] \\ \quad \texttt{\}} \\ \texttt{\}} \\ \texttt{else} \end{array} \right. ,$$

The last transformation rule is used for all decision subtrees with roots of the form $\alpha[sv_0, sv_1, \ldots]$, where the sv_i denote components of a constant shape vector:

$$\mathcal{C}_{Wrap}[\![\; i\,,\;\; \alpha[sv_0, sv_1, \ldots] : \mathit{Funs}, \mathit{Nextarg} \;]\!]$$

$$\longmapsto \left\{ \begin{array}{l} \texttt{if(shape(}a_i\texttt{) == [}sv_0, sv_1, \ldots\texttt{]) \{} \\ \quad \mathcal{C}_{NextArg}[\![\; i\,,\; \mathit{Funs}\,,\; \mathit{Nextarg} \;]\!] \\ \texttt{\}} \\ \texttt{else} \end{array} \right. .$$

The second transformation scheme $\mathcal{C}_{NextArg}$ is defined as follows:

$$\mathcal{C}_{NextArg}[\![\; i\,,\; \mathit{Funs}\,,\; \mathit{Nextarg} \;]\!]$$

$$\longmapsto \left\{ \begin{array}{lcr} b_1\texttt{, }\ldots\texttt{, }b_n\texttt{ = ERROR("type error");} & ; & \text{if } (\mathit{Nextarg} = ./.) \\ & & \wedge\, (\mathit{Funs} = \emptyset) \\ \hline b_1\texttt{, }\ldots\texttt{, }b_n\texttt{ = }f^{(k)}\texttt{(}a_1\texttt{, }\ldots\texttt{, }a_m\texttt{);} & & \\ \quad \text{where } (\langle k, d\rangle \in \mathit{Funs}) \wedge & ; & \text{if } (\mathit{Nextarg} = ./.) \\ \qquad\qquad (\forall_{\langle k', d'\rangle \in \mathit{Funs}} : d \leq d') & & \wedge\, (\mathit{Funs} \neq \emptyset) \\ \hline \mathcal{C}_{Wrap}[\![\; i{+}1\,, \mathit{Nextarg} \;]\!] & ; & \text{if } (\mathit{Nextarg} \neq ./.) \end{array} \right. .$$

If the label $\mathit{Nextarg}$ denotes an empty tree, the set Funs is searched for the index k with the smallest distance sum, in order to generate an application of the instance $f^{(k)}$. If no applicable instance of f exists, i. e., $\mathit{Funs} = \emptyset$, a code segment is generated which causes a runtime error. If $Nextarg$ denotes a non-empty tree, the transformation scheme $\mathcal{C}_{Wrap}$ is used recursively to generate the wrapper code for the next argument.

Applying the transformation scheme $\mathcal{C}_{Wrap}$ to the decision tree for the function `Det` (see Figure 4.4) yields a function body as given in Figure 4.1 (lines 8 – 21). The wrapper code obtained for the function `Foo` is shown in Figure 4.7.

4.3 Code Generation

Code generation for generic SAC programs is a rather complex task. Section 4.3.1 gives a new array representation that is suitable for the entire hierarchy of array types. However, for performance reasons, this representation consists not only of two different forms (one for scalars and one for non-scalars) but of four (one for each type category). This has only little impact on the ICM code used for the compilation rules (see Section 4.3.2), but implementing the ICMs itself is much more complicated than in the non-generic case. In order to support the various categories of array types, ICM definitions now have a tree structure, which is exemplified in Section 4.3.3.

4.3.1 Array Representation

Arrays are uniquely defined by means of a shape and a data vector. In general, size and contents of both vectors are unknown at compile time. Again, an additional reference counter (short: rc) for the implicit memory management is needed.

In order to get a compact representation, the reference counter and the shape vector are combined to a so-called *descriptor* containing reference counter, dimension, and all shape components. Keeping descriptor and data vector separately allows arrays to be handled uniquely irrespective of the length of their shape and data vectors, and facilitates interfacing to external languages such as C.

Unfortunately, this simple and uniform array representation does not suffice to obtain best possible runtime performance. Storing the shape in the descriptor (only) is in many situations inefficient, because the shape information is frequently used. For instance, consider a variable A representing an array of shape $[\bullet,\bullet]$. In this case it is guaranteed that all descriptors of the arrays, which

```
if (dim( a1) == 1) {
  if (shape( a1) == [2]) {
    /* nextarg */
    if (dim( a2) == 1) {
      if (shape( a2) == [3]) {
        ret = Foo__i_2_3( a1, a2);
      }
      else {
        ret = Foo__i_X_X( a1, a2);
      }
    }
    else {
      ret = ERROR( "type error");
    }
  }
  else {
    /* nextarg */
    if (dim( a2) == 1) {
      ret = Foo__i_X_X( a1, a2);
    }
    else {
      ret = ERROR( "type error");
    }
  }
}
else {
  /* nextarg */
  ret = ERROR( "type error");
}
```

Figure 4.7: Wrapper code for the function `Foo`.

are represented by A during program execution, contain a dimension of 2. So, in order to avoid costly accesses to the main memory, all references to the dimension of A should be replaced by the constant value 2. As a consequence, it is superfluous to store the dimension of A in the descriptor at all. Moreover, even the shape components of A are constant until a new array is assigned to A. Therefore, it is recommended to buffer the shape components on the runtime stack or even in registers.

But, it is unlikely that the C compiler will be able to apply such optimizations. In an imperative language like C any function call or any reference to a vector may cause a side-effect, hence it is almost impossible to detect that a value behind a pointer is constant or in fact superfluous. Therefore, rather than relying on the C compiler, these optimizations have to be done on the SAC level. For this purpose additional local variables are used, which always mirror the shape information of the descriptor. Whenever the shape has to be inspected, these local variables rather than the descriptor are accessed.

Figure 4.8 depicts the optimized C representations for the different categories of SAC arrays. For a variable A representing a non-scalar array, a descriptor A`_desc` and a data vector A`_data` are needed. The comment (`/*...*/`) behind the descriptor lists all its elements. Crossed out elements (e. g., ~~`dim`~~) denote statically known parts of the shape that are not expected in the descriptor (i. e., the corresponding descriptor entries possibly contain undefined values) and that are declared as scalar constants instead. Furthermore, all variable parts of the shape are mirrored in scalar variables. Note that mirroring the shape components is impossible for arrays of shape [+] or [*], because the number of needed scalars is unknown during compilation.

So, the hierarchy of array types in SAC is represented by a hierarchy of C representations. As a result, the compilation rules for array operations must be parameterized with respect to array categories, and in certain situations arrays must be converted from one representation into another.

4.3.2 Compilation Rules

With respect to ICM code most compilation rules can be more or less left as in the preceding chapter. However, some modifications are necessary. Since array shapes are not always statically known, the macro `ALLOC` needs a second argument sv that specifies how to compute the actual shape vector. Therefore, all rules which create an array must be revisited. Moreover, the compiler should optionally add runtime checks whenever type constraints could not be verified

Decl. in SAC	**Declaration in C**
$\alpha[\,]\ A$;	$\alpha\ A$;

$\alpha[4,3]\ A$;	const int A_dim = 2;
	const int A_size = 12;
	const int A_sv0 = 4;
	const int A_sv1 = 3;
	int *A_desc; /* rc, ~~dim~~, ~~size~~, ~~sv0~~, ~~sv1~~ */
	α *A_data;

$\alpha[\bullet,\bullet]\ A$;	const int A_dim = 2; int A_size; int A_sv0; int A_sv1; int *A_desc; /* rc, ~~dim~~, size, sv0, sv1 */ α *A_data;
$\alpha[+]\ A$; and $\alpha[*]\ A$;	int A_dim; int A_size; int *A_desc; /* rc, dim, size, sv0, ... */ α *A_data;

Figure 4.8: C *representations for the different categories of* SAC *arrays.*

at compile time. Only the rule for function applications, the rule for the primitive operation `reshape`, and the rule for `genarray`-`with`-loops need further changes.

The revised compilation rules for the basic language constructs are defined as follows. (For reasons of clarity, function definitions and applications are again restricted to a single argument and a single return value here. Nevertheless, the SAC compiler can handle arbitrary numbers of arguments and return values of course.) Function definitions are translated into the same ICM code as given in Section 3.5.2:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\sigma\ \ fun\texttt{(}\ \mathcal{T}\ \ A\texttt{)}\\ \texttt{\{}\\ \quad Vardecs\\ \quad Body\\ \quad \texttt{return(}\ B\!:\!\sigma\texttt{);}\\ \texttt{\}}\\ Rest\end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l}\texttt{FUN\_DEC(}\ fun\texttt{,}\\ \qquad\qquad \texttt{DEC\_OUT(}\ B\!:\!\sigma\texttt{),}\\ \qquad\qquad \texttt{DEC\_IN(}\ A\!:\!\mathcal{T}\texttt{))}\\ \texttt{\{}\\ \quad \mathcal{C}[\![\ Vardecs\ ]\!]\\ \quad \texttt{DECL\_SHAPE\_ARG(}\ A\!:\!\mathcal{T}\texttt{)}\\ \quad \texttt{ADJUST\_RC(}\ A\!:\!\mathcal{T}\texttt{,}\ \mathrm{Refs}(A)\!-\!1\texttt{)}\\ \quad \mathcal{C}[\![\ Body\ ]\!]\\ \quad \texttt{FUN\_RET(}\ B\!:\!\sigma\texttt{)}\\ \texttt{\}}\\ \mathcal{C}[\![\ Rest\ ]\!]\end{array}\right. .$$

However, the implementation of the macros `DEC_IN`, `DEC_OUT` as well as `DECL_SHAPE_ARG` must be modified to meet the new array representation (see Figure 4.8 on the page before). Consider, for example, that $\mathcal{T}$ denotes the type `float[.,.]`. That being the case, the macro `DEC_IN` would stand for

$$\texttt{int *}A\texttt{\_desc}\ \ ,\ \ \texttt{float *}A\texttt{\_data} \qquad ,$$

`DEC_OUT` would expand to the two reference parameters

$$\texttt{int **ret\_}B\texttt{\_desc}\ \ ,\ \ \beta\ \texttt{**ret\_}B\texttt{\_data} \qquad ,$$

and `DECL_SHAPE_ARG` would generate the following C declarations:

$$\begin{array}{l}\texttt{const int }A\texttt{\_dim = 2;}\\ \texttt{int }A\texttt{\_size;}\\ \texttt{int }A\texttt{\_sv0;}\\ \texttt{int }A\texttt{\_sv1;}\end{array} \qquad .$$

Variable declarations are also compiled the usual way:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\mathcal{T}\ \ A\texttt{;}\\ Rest\end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l}\texttt{DECL(}\ A\!:\!\mathcal{T}\texttt{)}\\ \mathcal{C}[\![\ Rest\ ]\!]\end{array}\right. .$$

Here, the macro DECL generates a C declaration as outlined in Figure 4.8 of course.

Likewise, no modifications are needed in the ICM code for assignments with a variable on the right hand side:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B:\sigma \texttt{ = } A:\tau\texttt{;} \\ \mathit{Rest} \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \texttt{ASSIGN(}\ B:\sigma\texttt{,}\ A:\tau\texttt{)} \\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\ \mathcal{C}[\![\ \mathit{Rest}\ ]\!] \end{array}\right. .$$

But the macro ASSIGN has to convert the array representation if necessary. Take as an example $\sigma \equiv$ `float[+]` and $\tau \equiv$ `float[•,•]`. Then, the following C code is created for ASSIGN:

```
B_dim     = A_dim;      /* assign mirror */
B_size    = A_size;     /* assign mirror */
B_desc    = A_desc;     /* assign descriptor */
B_desc[1] = A_dim;      /* adjust descriptor */
B_data    = A_data;     /* assign data vector */       .
```

However, if the inferred types are $\sigma \equiv$ `float[•,•]` and $\tau \equiv$ `float[+]`, a runtime check[2] (RT_CHECK) is added to assure that A is a two-dimensional array:

```
RT_CHECK( A_dim = 2)    /* check dimension */
B_size = A_size;        /* assign mirror */
B_sv0  = A_desc[3];     /* assign mirror */
B_sv1  = A_desc[4];     /* assign mirror */
B_desc = A_desc;        /* assign descriptor */
B_data = A_data;        /* assign data vector */       .
```

Another important situation arises if B has shape $[*]$ and A has shape $[\,]$, or vice versa. If $\sigma \equiv$ `float[*]` and $\tau \equiv$ `float[]`, memory for B has to be allocated:

```
ALLOC( B:σ, [])
B_data[0] = A;          /* assign data vector */       .
```

If $\sigma \equiv$ `float[]` and $\tau \equiv$ `float[*]`, the data of A is *not* reused by B, thus, the reference counter of A must be decremented:

```
RT_CHECK( A_dim = 2)    /* check dimension */
RT_CHECK( A_size = 1)   /* check size */
B = A_data[0];          /* assign data vector */
ADJUST_RC( A:τ, −1)                                    .
```

Assignments with a vector construct of length m on the right hand side require the creation of a new array (ALLOC) whose shape equals the concatenation of $[m]$ and the shape of the vector elements. In order to guarantee that all

[2]Runtime checks are optional. For optimal runtime performance all runtime checks can be disabled by means of a compiler flag.

vector elements have identical shapes, additional runtime checks (RT_CHECK) are added. The rest of the ICM code is the same as given in Section 3.5.2:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}B:\sigma\ \texttt{=}\ \texttt{[}\ A_1:\mathcal{T}_1\texttt{,}\ \ldots\texttt{,}\ A_m:\mathcal{T}_m\ \texttt{];}\\ \mathit{Rest}\end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{ll}\texttt{RT\_CHECK(}\ \texttt{SV(}\,A_i:\mathcal{T}_i\texttt{)}\ =\ \texttt{SV(}\,A_1:\mathcal{T}_1\texttt{))} & ,\quad i\in\{2,\ldots,m\}\\ \texttt{ALLOC(}\ B:\sigma\texttt{,}\ [m]+\!\!+\,\texttt{SV(}\,A_1:\mathcal{T}_1\texttt{))} & \\ \texttt{SET\_DV(}\ B:\sigma\texttt{,}\ \texttt{DV(}\,A_1:\mathcal{T}_1\texttt{)}+\!\!+\ldots+\!\!+\texttt{DV(}\,A_m:\mathcal{T}_m\texttt{))} & \\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} & \\ \texttt{ADJUST\_RC(}\ A_i:\mathcal{T}_i\texttt{,}\ -1\texttt{)} & ,\quad i\in\{1,\ldots,m\}\\ \mathcal{C}[\![\ \mathit{Rest}\ ]\!] & \end{array}\right. .$$

Assignments with a constant scalar on the right hand side are compiled similar to the preceding rule. However, computing the shape vector is trivial here — the shape equals the empty vector:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}B:\sigma\ \texttt{=}\ \mathit{val}\texttt{;}\\ \mathit{Rest}\end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l}\texttt{ALLOC(}\ B:\sigma\texttt{,}\ [\,]\texttt{)}\\ \texttt{SET\_DV(}\ B:\sigma\texttt{,}\ [\mathit{val}]\texttt{)}\\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)}\\ \mathcal{C}[\![\ \mathit{Rest}\ ]\!]\end{array}\right. .$$

Assignments with a function application are compiled as follows:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}B:\sigma\ \texttt{=}\ \mathit{fun}\texttt{(}\ A:\mathcal{T}\texttt{);}\\ \mathit{Rest}\end{array}\right]\!\!\right] \quad \text{with function definition} \quad \sigma'\ \mathit{fun}\texttt{(}\,\mathcal{T}'\ A'\texttt{)}\ \texttt{\{}\ldots\texttt{\}}$$

$$\longmapsto \left\{\begin{array}{ll}\mathcal{C}\left[\!\!\left[\begin{array}{l}A':\mathcal{T}'\ \texttt{=}\ A:\mathcal{T}\texttt{;}\\ B:\sigma\ \texttt{=}\ \mathit{fun}\texttt{(}\ A':\mathcal{T}'\texttt{);}\\ \mathit{Rest}\end{array}\right]\!\!\right] & ;\ \text{if } \mathcal{T}\neq\mathcal{T}'\\ \hline \mathcal{C}\left[\!\!\left[\begin{array}{l}B':\sigma'\ \texttt{=}\ \mathit{fun}\texttt{(}\ A:\mathcal{T}\texttt{);}\\ B:\sigma\ \texttt{=}\ B':\sigma'\texttt{;}\\ \mathit{Rest}\end{array}\right]\!\!\right] & ;\ \text{if } \sigma\neq\sigma'\\ \hline \begin{array}{l}\texttt{FUN\_AP(}\ \mathit{fun}\texttt{,}\\ \qquad\texttt{AP\_OUT(}\ B:\sigma\texttt{),}\\ \qquad\texttt{AP\_IN(}\ A:\mathcal{T}\texttt{))}\\ \texttt{REFRESH\_MIRROR(}\ B:\sigma\texttt{)}\\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)}\\ \mathcal{C}[\![\ \mathit{Rest}\ ]\!]\end{array} & ;\ \text{otherwise}\end{array}\right. .$$

If the types of formal and actual arguments / return values are identical, the assignment is directly transformed into the two statements FUN_AP and REFRESH_MIRROR. Otherwise additional assignments before or after the function application are inserted to convert the array representations accordingly.

Suppose $\mathcal{T} = \mathcal{T}'$ and $\sigma = \sigma'$ to be both non-scalar types. Similar to the rule given in Section 3.5.2, the macro `FUN_AP` expands to an application of the function fun to the arguments

$$\begin{array}{llll} \&B\texttt{\_desc} & , & \&B\texttt{\_data} & , \\ A\texttt{\_desc} & , & A\texttt{\_data} & . \end{array}$$

Note that the mirror variables of the array representation (B`_dim`, etc.) are *not* passed to the function, because the function signature must be compatible with all argument types. Instead, the subsequent statement `REFRESH_MIRROR` assures that the mirror variables are initialized with the corresponding values of the descriptor. If, for instance, σ represents the type `float`$[\bullet,\bullet]$, the following assignments are required:

$$\begin{array}{lcl} B\texttt{\_size} & = & B\texttt{\_desc[2];} \\ B\texttt{\_sv0} & = & B\texttt{\_desc[3];} \\ B\texttt{\_sv1} & = & B\texttt{\_desc[4];} \end{array} \quad .$$

As shown in the preceding chapter, compiling the primitive operation `reshape` in a non-generic setting with statically known shape vectors is trivial. Since the operation does not affect the data vector at all, it can be handled like a copy assignment. With dynamic shape information being part of the array representation this simplification is no longer applicable in general:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B:\sigma \texttt{ = reshape(} sv:\mathcal{T}'\texttt{, } A:\mathcal{T}\texttt{);} \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{l} \texttt{if(} A\texttt{\_desc[0]} > 1\texttt{) \{} \quad \texttt{/* rc */} \\ \quad \texttt{ALLOC(} B:\sigma\texttt{, DV(} sv:\mathcal{T}'\texttt{))} \\ \quad \texttt{RT\_CHECK(} B\texttt{\_size} = A\texttt{\_size)} \\ \quad \texttt{SET\_DV(} B:\sigma\texttt{, DV(} A:\mathcal{T}\texttt{))} \\ \quad \texttt{ADJUST\_RC(} A:\mathcal{T}\texttt{, } -1\texttt{)} \\ \texttt{\}} \\ \texttt{else \{} \\ \quad \texttt{ASSIGN(} B:\sigma\texttt{, } A:\mathcal{T}\texttt{)} \\ \texttt{\}} \\ \texttt{ADJUST\_RC(} B:\sigma\texttt{, } \mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(} sv:\mathcal{T}'\texttt{, } -1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. \quad .$$

Assuming that the reference counter of the source array is greater than 1, A and B should not be considered the same array because the contents of their descriptors may differ (or may get different during further program execution). Take as an example the following fragment of a SAC program:

```
int[2,3] A;
int[3,2] B;
int[.,.] C;
A = ...;
B = reshape( [3,2], A);
C = B;
```
.

After the `reshape` operation, the descriptors of `A` and `B` both contain the reference counter only. But after the subsequent assignment `C = B;` , the descriptor of `B` (alias `C`) is filled with shape information that would be inappropriate for `A`. As a consequence, `reshape` has to create a new array (`ALLOC`) and must copy the whole data vector (`SET_DV`). Copying the data vector could be avoided here, however, by using an array representation with separate reference counters for data and shape vector, which would allow to create a new shape vector while reusing the data vector. But, `reshape` operations on non-reusable arrays are very rare in practice, and using two reference counters would double the reference counting overhead for *all* array operations. Therefore, it is not recommended to use such an inefficient array representation.

The ICM code for the other primitive array operations (including `modarray`) is basically the same as in the old compilation scheme, the only differences being the modified signature of the `ALLOC` macro and some additional runtime checks. The `modarray` operation is compiled as follows:

$$
\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\mathcal{O}\ \ \texttt{= modarray(}\ A\!:\!\mathcal{T}\texttt{,}\ \ iv\!:\!\mathcal{T}'\texttt{,}\ \ val\!:\!\mathcal{T}''\texttt{);} \\ Rest \end{array}\right]\!\!\right]
$$

$$
\longmapsto \left\{\begin{array}{l}
\texttt{RT\_CHECK(}\ iv\texttt{\_dim} \ = \ 1\texttt{)} \\
\texttt{RT\_CHECK(}\ iv\texttt{\_size} \ \leq \ A\texttt{\_dim)} \\
\texttt{RT\_CHECK(SV(}\,val\!:\!\mathcal{T}''\texttt{)} \ = \ \texttt{SV}(A\!:\!\mathcal{T})\big|_{iv\texttt{\_size}}^{A\texttt{\_dim}-1}\texttt{)} \\
\texttt{if(}\ A\texttt{\_desc[0]} > 1\texttt{) \{}\quad \texttt{/* rc */} \\
\quad \texttt{ALLOC(}\ B\!:\!\mathcal{O}\texttt{, SV(}\,A\!:\!\mathcal{T}\texttt{))} \\
\quad \texttt{SET\_DV\_PRF(}\ B\!:\!\mathcal{O}\texttt{, modarray,}\ A\!:\!\mathcal{T}\texttt{,}\ iv\!:\!\mathcal{T}'\texttt{,}\ val\!:\!\mathcal{T}''\texttt{)} \\
\quad \texttt{ADJUST\_RC(}\ A\!:\!\mathcal{T}\texttt{,}\ -1\texttt{)} \\
\texttt{\}} \\
\texttt{else \{} \\
\quad \texttt{ASSIGN(}\ B\!:\!\mathcal{O}\texttt{,}\ A\!:\!\mathcal{T}\texttt{)} \\
\quad \texttt{SET\_DV\_SUB(}\ B\!:\!\mathcal{O}\texttt{,}\ iv\!:\!\mathcal{T}'\texttt{, DV(}\,val\!:\!\mathcal{T}''\texttt{))} \\
\texttt{\}} \\
\texttt{ADJUST\_RC(}\ B\!:\!\mathcal{O}\texttt{,}\ \mathrm{Refs}(B)-1\texttt{)} \\
\texttt{ADJUST\_RC(}\ iv\!:\!\mathcal{T}'\texttt{,}\ -1\texttt{)} \\
\texttt{ADJUST\_RC(}\ val\!:\!\mathcal{T}''\texttt{,}\ -1\texttt{)} \\
\mathcal{C}\,[\![\ Rest\ ]\!]
\end{array}\right.
$$
,

where the runtime checks guarantee that iv is a legal index vector, and that val has the correct size with respect to A. (The formula $[v_0, v_1, v_2, \ldots]\big|_j^k$ denotes the vector $[v_j, v_{j+1}, \ldots, v_k]$.)

The compilation rules for the remaining primitive array operations have the following form:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\sigma \;\texttt{=}\; prf\texttt{(}\; A_1\!:\!\mathcal{T}_1\texttt{,}\;\ldots\texttt{,}\; A_m\!:\!\mathcal{T}_m\texttt{);} \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{l} \texttt{RT\_CHECK(}\;\ldots\texttt{)} \\ \texttt{ALLOC(}\; B\!:\!\sigma\texttt{,} \\ \qquad\quad \texttt{CALC\_SV\_PRF(}\; B\!:\!\sigma\texttt{,}\; prf\texttt{,}\; A_1\!:\!\mathcal{T}_1\texttt{,}\;\ldots\texttt{,}\; A_m\!:\!\mathcal{T}_m\texttt{))} \\ \texttt{SET\_DV\_PRF(}\; B\!:\!\sigma\texttt{,}\; prf\texttt{,}\; A_1\!:\!\mathcal{T}_1\texttt{,}\;\ldots\texttt{,}\; A_m\!:\!\mathcal{T}_m\texttt{)} \\ \texttt{ADJUST\_RC(}\; B\!:\!\sigma\texttt{,}\; \mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(}\; A_i\!:\!\mathcal{T}_i\texttt{,}\; -1\texttt{)} \qquad , \qquad i \in \{1,\ldots,m\} \\ \mathcal{C}[\![\; Rest \;]\!] \end{array}\right. ,$$

where the statement `CALC_SV_PRF` represents the prf-specific computation of the shape vector for the result.

The compilation rule for `fold`-`with`-loops can be left as given in the preceding chapter:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\sigma \;\texttt{=}\; \texttt{with(}\;\ldots\; iv\!:\!\mathcal{T}'\;\ldots\texttt{)}\;\texttt{\{} \\ \qquad\qquad expr\!:\!\mathcal{T}'' \;\texttt{=}\;\ldots\; iv\!:\!\mathcal{T}'\;\ldots\texttt{;} \\ \qquad\quad \texttt{\}}\;\texttt{fold(}\; foldop\texttt{,}\; neutral\!:\!\mathcal{T}\texttt{,}\; expr\!:\!\mathcal{T}''\texttt{);} \\ \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{l} \mathcal{C}[\![\; B\!:\!\sigma \;\texttt{=}\; neutral\!:\!\mathcal{T}\texttt{;}\;]\!] \\ \texttt{WITH\_LOOP(}\; iv\texttt{,} \\ \quad \mathcal{C}[\![\; expr\!:\!\mathcal{T}'' \;\texttt{=}\;\ldots\; iv\!:\!\mathcal{T}'\;\ldots\texttt{;}\;]\!] \\ \quad \mathcal{C}[\![\; B\!:\!\sigma \;\texttt{=}\; foldop\texttt{(}\; B\!:\!\sigma, expr\!:\!\mathcal{T}''\texttt{);}\;]\!] \\ \texttt{)} \\ \texttt{ADJUST\_RC(}\; B\!:\!\sigma\texttt{,}\; \mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(}\; neutral\!:\!\mathcal{T}\texttt{,}\; -1\texttt{)} \\ \mathcal{C}[\![\; Rest \;]\!] \end{array}\right. .$$

However, the implementation of the macro `WITH_LOOP` is much more complex here. If the generator of the `with`-loop specifies index vectors with unknown shape, the dimension of the iteration space is unknown at compile time. Thus, it is impossible to create a static loop nesting with separate loops for each axis. By contrast, the proper index vector has to be computed for each iteration step dynamically. Consider as an example the following `with`-loop:

```
with( a <= iv < b)
fold( ...)                                    .
```

If a and b are statically identified as vectors with two elements, the iteration over the index vectors specified by the generator could be implemented like this:

```
for( iv[0] = a[0]; iv[0] < b[0]; iv[0]++) {
  for( iv[1] = a[1]; iv[1] < b[1]; iv[1]++) {
    ...
  }
}                                             .
```

But without static knowledge of the shape, a dynamic loop with weaker runtime performance is needed:

```
iv = a;
while( iv < b) {
  ...
  iv = compute_next_index( iv, a, b);
}                                             ,
```

where the loop condition (iv < b) and the function compute_next_index both operate on vectors rather than on scalars.

The next rule applies to modarray-with-loops:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\sigma \;\texttt{= with(}\;\ldots\; iv\!:\!\mathcal{T}'\;\ldots\texttt{) \{} \\ \qquad\qquad expr\!:\!\mathcal{T}'' \;\texttt{=}\;\ldots\; iv\!:\!\mathcal{T}'\;\ldots\texttt{;} \\ \qquad\quad \texttt{\} modarray(}\; A\!:\!\mathcal{T}\texttt{,}\; iv\!:\!\mathcal{T}'\texttt{,}\; expr\!:\!\mathcal{T}''\texttt{);} \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{l} \texttt{ALLOC(}\; B\!:\!\sigma\texttt{, SV(}\; A\!:\!\mathcal{T}\texttt{))} \\ \texttt{WITH\_LOOP(}\; iv\texttt{,} \\ \quad \mathcal{C}[\![\; expr\!:\!\mathcal{T}'' \;\texttt{=}\;\ldots\; iv\!:\!\mathcal{T}'\;\ldots\texttt{;}\;]\!] \\ \quad \texttt{RT\_CHECK(SV(}\, expr\!:\!\mathcal{T}''\texttt{)} \;=\; \texttt{SV(}B\!:\!\sigma\texttt{)}\big|_{iv\texttt{\_size}}^{B\texttt{\_dim}-1}\texttt{)} \\ \quad \texttt{SET\_DV\_SUB(}\; B\!:\!\sigma\texttt{,}\; iv\texttt{, DV(}\, expr\!:\!\mathcal{T}''\texttt{))} \\ \quad \texttt{ADJUST\_RC(}\; expr\!:\!\mathcal{T}''\texttt{,}\; -1\texttt{)} \\ \texttt{,} \\ \quad \texttt{COPY\_DV\_SUB(}\; B\!:\!\sigma\texttt{,}\; iv\texttt{,}\; A\!:\!\mathcal{T}\texttt{)} \\ \texttt{)} \\ \texttt{ADJUST\_RC(}\; B\!:\!\sigma\texttt{,}\; \mathrm{Refs}(B)-1\texttt{)} \\ \texttt{ADJUST\_RC(}\; A\!:\!\mathcal{T}\texttt{,}\; -1\texttt{)} \\ \mathcal{C}[\![\; Rest\;]\!] \end{array}\right. .$$

Apart from the modified signature of the `ALLOC` macro and an added `RT_CHECK` statement, this is the same ICM code as given in the preceding chapter.[3] The runtime check assures that the instances of $expr$ have identical shapes in each iteration step and fit into the result B.

By contrast, compiling `genarray`-`with`-loops is not that easy. With the presence of dynamic shapes, the rule of the old compilation scheme is not generally applicable anymore:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\sigma \;\texttt{= with(}\;\ldots\;\; iv\!:\!\mathcal{T}'\;\;\ldots\texttt{) \{} \\ \qquad\qquad expr\!:\!\mathcal{T}''\;\texttt{=}\;\ldots\;\; iv\!:\!\mathcal{T}'\;\;\ldots\texttt{;} \\ \qquad\quad \texttt{\} genarray(}\; sv\!:\!\mathcal{T}\texttt{,}\;\; expr\!:\!\mathcal{T}''\texttt{);} \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{l} \texttt{ALLOC(}\;B\!:\!\sigma\texttt{,}\;\texttt{DV(}\;sv\!:\!\mathcal{T}\texttt{)} \mathbin{+\!\!+} \texttt{SV(}\;expr\!:\!\mathcal{T}''\texttt{))} \\ \texttt{WITH\_LOOP(}\;iv\texttt{,} \\ \quad \mathcal{C}[\![\; expr\!:\!\mathcal{T}''\;\texttt{=}\;\ldots\;\; iv\!:\!\mathcal{T}'\;\;\ldots\texttt{;}\;]\!] \\ \quad \texttt{SET\_DV\_SUB(}\;B\!:\!\sigma\texttt{,}\;\; iv\texttt{,}\;\;\texttt{DV(}\;expr\!:\!\mathcal{T}''\texttt{))} \\ \quad \texttt{ADJUST\_RC(}\;expr\!:\!\mathcal{T}''\texttt{,}\;\;-1\texttt{)} \\ \texttt{,} \\ \quad \texttt{SET\_DV\_SUB(}\;B\!:\!\sigma\texttt{,}\;\; iv\texttt{,}\;\;[0,\ldots,0]\texttt{)} \\ \texttt{)} \\ \texttt{ADJUST\_RC(}\;B\!:\!\sigma\texttt{,}\;\;\mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(}\;sv\!:\!\mathcal{T}\texttt{,}\;\;-1\texttt{)} \\ \mathcal{C}[\![\; Rest \;]\!] \end{array}\right. .$$

The problem is that the macro `ALLOC` needs instructions how to compute the shape of the result B. The semantics of SAC prescribes that the shape equals

$$\texttt{DV(}\;sv\!:\!\mathcal{T}\texttt{)} \;\mathbin{+\!\!+}\; \texttt{SV(}\;expr\!:\!\mathcal{T}''\texttt{)} \qquad ,$$

hence, the shape of B depends on the shape of $expr$. But $expr$ is computed *inside* the loop only, therefore, during execution of the `ALLOC` instruction, $expr$ is still undefined. For that reason, the compilation rule given above is applicable only if the shape of $expr$ has been inferred statically. In all other cases it would be necessary to postpone memory allocation for B until the first instance of $expr$ has been computed.

If the meaning of $expr$ does not depend on iv, there exists a simple solution for this problem. Since the computation of $expr$ is loop invariant, it can be

[3] Similar to the rule for the primitive operation `modarray`, the compiler may add branch conditions to reuse already allocated arrays rather than allocating new memory. For reasons of clarity, this has been left out here.

moved in front of the `with`-loop.[4] This leads to the following compilation rule:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B:\sigma\ \texttt{= with(...}\ iv:\mathcal{T}'\ \texttt{...) \{} \\ \qquad\qquad expr:\mathcal{T}''\ \texttt{= ...;} \\ \qquad\quad \texttt{\} genarray(}\ sv:\mathcal{T}\texttt{,}\ expr:\mathcal{T}''\texttt{);} \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{l} \mathcal{C}[\![\ expr:\mathcal{T}''\ \texttt{= ...;}\ ]\!] \\ \texttt{ALLOC(}\ B:\sigma\texttt{, DV(}\ sv:\mathcal{T}\texttt{)} + \!\!\! + \texttt{SV(}\ expr:\mathcal{T}''\texttt{))} \\ \texttt{WITH\_LOOP(}\ iv\texttt{,} \\ \quad \texttt{SET\_DV\_SUB(}\ B:\sigma\texttt{,}\ iv\texttt{, DV(}\ expr:\mathcal{T}''\texttt{))} \\ \texttt{,} \\ \quad \texttt{SET\_DV\_SUB(}\ B:\sigma\texttt{,}\ iv\texttt{,}\ [0,\ldots,0]\texttt{)} \\ \texttt{)} \\ \texttt{ADJUST\_RC(}\ B:\sigma\texttt{,}\ \mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(}\ sv:\mathcal{T}\texttt{,}\ -1\texttt{)} \\ \texttt{ADJUST\_RC(}\ expr:\mathcal{T}''\texttt{,}\ -1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array}\right. .$$

In general, however, the value of $expr$ will depend on the index variable iv. In such a situation, the `ALLOC` instruction must be executed after the first instance of $expr$ has been computed. The easiest way to tackle this problem is to perform *loop peeling* [BGS94], i. e., to extract the code for a single loop iteration (e. g., for the smallest index vector specified by the generator) and to move this code in front of the `with`-loop:

[4]In fact, this optimization is done already during loop invariant removal as part of the high-level code optimizations (see Section 3.3).

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\sigma \texttt{ = with(} a\!:\!\mathcal{T}' \texttt{ <= } iv\!:\!\mathcal{T}' \texttt{ <= } b\!:\!\mathcal{T}' \texttt{ ...) \{} \\ \qquad\qquad expr\!:\!\mathcal{T}'' \texttt{ = ... } iv\!:\!\mathcal{T}' \texttt{ ...;} \\ \qquad\quad \texttt{\} genarray(} sv\!:\!\mathcal{T}\texttt{, } expr\!:\!\mathcal{T}''\texttt{);} \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{l} \mathcal{C}\left[\!\!\left[\begin{array}{l} iv\!:\!\mathcal{T}' \texttt{ = } a\!:\!\mathcal{T}'\texttt{;} \\ expr\!:\!\mathcal{T}'' \texttt{ = ... } iv\!:\!\mathcal{T}' \texttt{ ...;} \end{array}\right]\!\!\right] \\ \texttt{ALLOC(} B\!:\!\sigma\texttt{, DV(} sv\!:\!\mathcal{T}\texttt{)} +\!\!\!+ \texttt{SV(} expr\!:\!\mathcal{T}''\texttt{))} \\ \texttt{SET\_DV\_SUB(} B\!:\!\sigma\texttt{, } iv\texttt{, DV(} expr\!:\!\mathcal{T}''\texttt{))} \\ \texttt{ADJUST\_RC(} expr\!:\!\mathcal{T}''\texttt{, } -1\texttt{)} \\ \texttt{WITH\_LOOP'(} iv\texttt{,} \\ \quad \mathcal{C}[\![\; expr\!:\!\mathcal{T}'' \texttt{ = ... } iv\!:\!\mathcal{T}' \texttt{ ...; }]\!] \\ \quad \texttt{RT\_CHECK(SV(} expr\!:\!\mathcal{T}''\texttt{)} = \texttt{SV}(B\!:\!\sigma)\big|_{iv\texttt{\_size}}^{B\texttt{\_dim}-1}\texttt{)} \\ \quad \texttt{SET\_DV\_SUB(} B\!:\!\sigma\texttt{, } iv\texttt{, DV(} expr\!:\!\mathcal{T}''\texttt{))} \\ \quad \texttt{ADJUST\_RC(} expr\!:\!\mathcal{T}''\texttt{, } -1\texttt{)} \\ \texttt{,} \\ \quad \texttt{SET\_DV\_SUB(} B\!:\!\sigma\texttt{, } iv\texttt{, } [0,\ldots,0]\texttt{)} \\ \texttt{)} \\ \texttt{ADJUST\_RC(} B\!:\!\sigma\texttt{, } \mathrm{Refs}(B){-}1\texttt{)} \\ \texttt{ADJUST\_RC(} sv\!:\!\mathcal{T}\texttt{, } -1\texttt{)} \\ \mathcal{C}[\![\; Rest \;]\!] \end{array}\right. .$$

The assignments in front of the `ALLOC` instruction compute the value of $expr$ for the smallest instance of the index vector iv which is named a here. The subsequent `SET_DV_SUB` statement stores this value in the array B. The rest of the ICM code is constructed in the familiar way, except that the macro `WITH_LOOP` has been replaced by `WITH_LOOP'` which indicates that one index vector (a) has been removed from the iteration space.

Unfortunately, even this revised compilation rule fails if the generator of the `with`-loop specifies an *empty* index vector set, i. e., it is $(a \not\leq b)$. Since the domain of the index variable iv is empty, no instance of $expr$ exists. Hence, it is impossible to use $expr$ to calculate the shape of B in this situation, and the compiler must return an error message. If the condition $(a \leq b)$ can not be evaluated statically, this check must be done at runtime:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} B\!:\!\sigma \texttt{ = with(} a\!:\!\mathcal{T}' \texttt{ <= } iv\!:\!\mathcal{T}' \texttt{ <= } b\!:\!\mathcal{T}' \texttt{ ...) \{} \\ \qquad\qquad expr\!:\!\mathcal{T}'' \texttt{ = ... } iv\!:\!\mathcal{T}' \texttt{ ...;} \\ \qquad\quad \texttt{\} genarray(} sv\!:\!\mathcal{T}\texttt{, } expr\!:\!\mathcal{T}''\texttt{);} \\ Rest \end{array}\right]\!\!\right]$$

$$\longmapsto \left\{ \begin{array}{l} \texttt{if(} a \leq b \texttt{) \{} \\ \quad \mathcal{C}\left[\!\!\left[\begin{array}{l} iv\!:\!\mathcal{T}' \texttt{ = } a\!:\!\mathcal{T}'\texttt{;} \\ expr\!:\!\mathcal{T}'' \texttt{ = ... } iv\!:\!\mathcal{T}' \texttt{ ...;} \end{array}\right]\!\!\right] \\ \quad \texttt{ALLOC(} B\!:\!\sigma\texttt{, DV(} sv\!:\!\mathcal{T}\texttt{)} +\!\!\!+ \texttt{SV(} expr\!:\!\mathcal{T}''\texttt{))} \\ \quad \texttt{SET\_DV\_SUB(} B\!:\!\sigma\texttt{, } iv\texttt{, DV(} expr\!:\!\mathcal{T}''\texttt{))} \\ \quad \texttt{ADJUST\_RC(} expr\!:\!\mathcal{T}''\texttt{, } -1\texttt{)} \\ \texttt{\}} \\ \texttt{else \{} \\ \quad \texttt{RT\_ERROR("Result of WL has unknown shape")} \\ \texttt{\}} \\ \texttt{WITH\_LOOP'(} iv\texttt{,} \\ \quad \mathcal{C}[\![\ expr\!:\!\mathcal{T}'' \texttt{ = ... } iv\!:\!\mathcal{T}' \texttt{ ...; }]\!] \\ \quad \texttt{RT\_CHECK(SV(} expr\!:\!\mathcal{T}''\texttt{)} = \texttt{SV(}B\!:\!\sigma\texttt{)}\Big|_{iv\texttt{\_size}}^{B\texttt{\_dim}-1}\texttt{)} \\ \quad \texttt{SET\_DV\_SUB(} B\!:\!\sigma\texttt{, } iv\texttt{, DV(} expr\!:\!\mathcal{T}''\texttt{))} \\ \quad \texttt{ADJUST\_RC(} expr\!:\!\mathcal{T}''\texttt{, } -1\texttt{)} \\ \texttt{,} \\ \quad \texttt{SET\_DV\_SUB(} B\!:\!\sigma\texttt{, } iv\texttt{, } [0,\ldots,0]\texttt{)} \\ \texttt{)} \\ \texttt{ADJUST\_RC(} B\!:\!\sigma\texttt{, } \mathrm{Refs}(B)-1\texttt{)} \\ \texttt{ADJUST\_RC(} sv\!:\!\mathcal{T}\texttt{, } -1\texttt{)} \\ \mathcal{C}[\![\ Rest\ ]\!] \end{array} \right. .$$

4.3.3 Intermediate Code Macros

The compilation rules defined in the preceding subsection do not produce pure C code, but C code enriched with intermediate code macros (ICMs). This is done to liberate the compilation scheme from a concrete implementation, for instance from the choice of the array representation.

In a non-generic setting as given in Chapter 3 these ICMs can be translated into plain C code rather straightforwardly. Since the shapes of all arrays are statically known, only two different array representations are needed: scalars and non-scalars. Moreover, there is no need to convert arrays from one representation into another. Hence, for each ICM exist at most two variants of implementation, one for scalar arguments and one for non-scalar arguments.

With the presence of dynamic shapes the situation is much more complicated though. To achieve utmost runtime performance, *four* different array representations are used. Furthermore, function applications may require arrays to be converted from one array representation into another, and array operations may be applied to arguments of manifold type configurations. As a consequence, for many ICMs exist a large number of different implementation variants for different argument types.

Take as an example the ICM

$$\texttt{ASSIGN(}\ B\texttt{:}\sigma\texttt{,}\ A\texttt{:}\tau\texttt{)} \qquad ,$$

which assigns the source array A to the target array B and converts the array representation accordingly. This macro must be implemented for 16 different combinations of argument types. (Two arguments with four different array representations each $\rightarrow 4^2 = 16$ combinations.) Some of these C implementations have already been given on page 65. But, in order to simplify compiler maintenance by reducing the number of implementation variants, it is recommended to use further ICM layers rather than generating C code directly.

On the lowest level of abstraction, several ICMs for accessing the shape information of an array (i. e., dimension, size, and shape components) are defined. The shape information of an array A may be located in the descriptor (A`_desc`) or in the mirror variables (A`_dim`, A`_size`, A`_sv0`, $\ldots$), therefore, two sets of access macros are needed with prefixes `DESC` and `MIRROR`. For read accesses, it is useful to define a third set of macros with prefix `READ` which expands to a mirror access whenever a suitable mirror is available and to a descriptor access otherwise. The definitions of all these macros are given in Figure 4.9. The boxes with thick frames contain the signatures of the ICMs, and the arrows point to the code fragments to be generated for them. If an ICM has multiple arrows, the code generation depends on predicates which are specified in a label next to the arrow.

With these ICM definitions at hand, shape information of an array can be read or modified without bothering with the concrete C representation. To take this idea a step further, it is useful to define another layer of ICMs. In all but one case, the macro `ASSIGN` has to carry out the following five tasks:

- Assure that the constant mirrors of B contain correct values with respect to A.

- Update the mirrors of B, i. e., assign all non-constant mirrors.

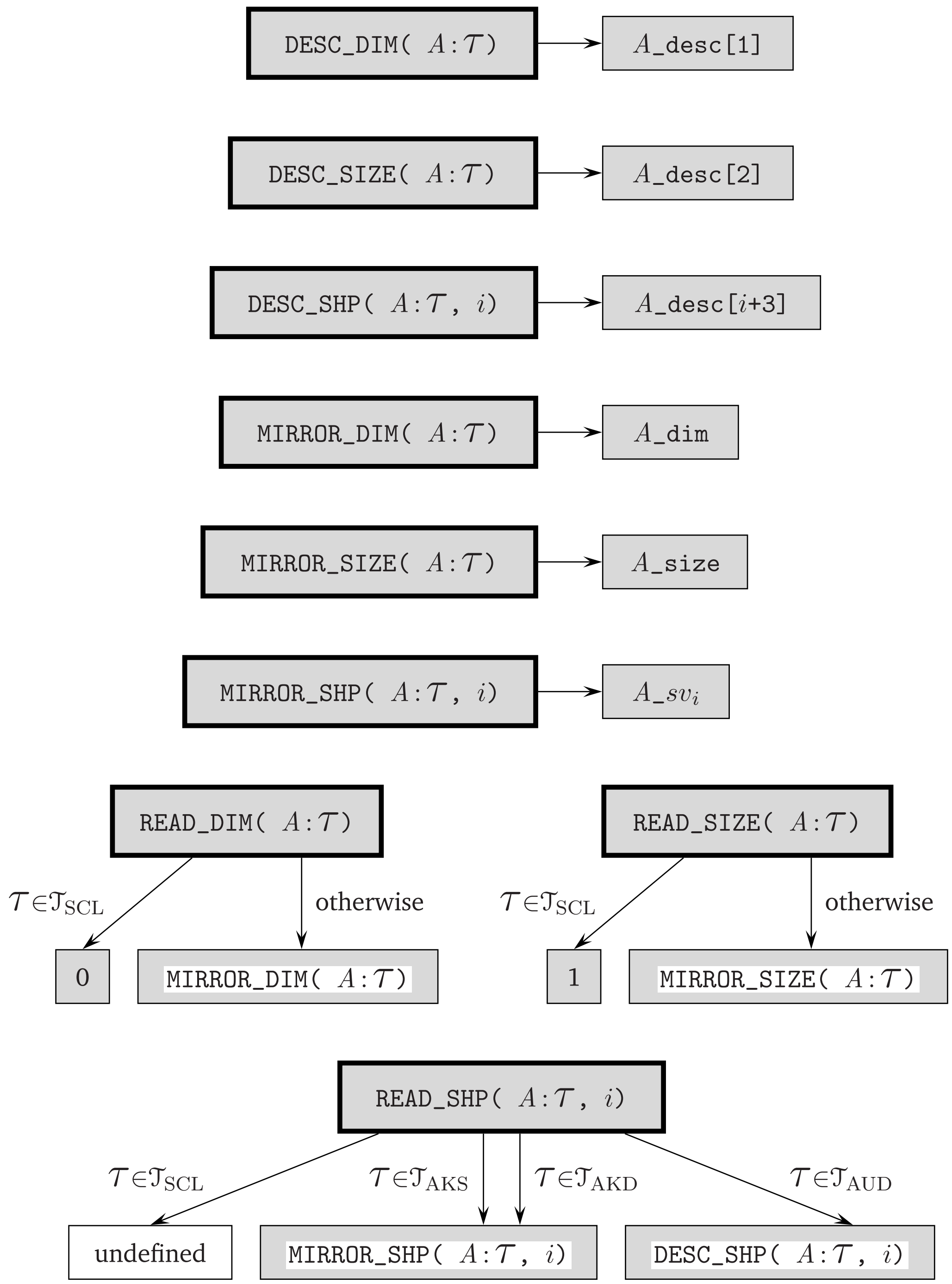


Figure 4.9: Intermediate code macros for accessing shape information.

- Assign the descriptor of B if present.
- Update the descriptor of B if present, i. e., assign all relevant descriptor entries.
- Assign the data vector of B.

If the source array A is a scalar, however, the situation is a bit different:

- Assure that the constant mirrors of B contain correct values with respect to A.
- Allocate memory for B, i. e., use the macro `ALLOC`.
- Assign the data vector of B.

Using the access macros defined in Figure 4.9, each of this tasks can be implemented rather straitforwardly as a separate ICM which is demonstrated in Figures 4.10 and 4.11.

This hierarchical definition of the `ASSIGN` macro has several advantages over a monolithic definition. First of all, the number of implementation variants is smaller — no ICM requires more than 4 variants rather than 16. Moreover, most of the additionally defined macros can be reused for other ICM implementations which reduces the implementation effort and eases compiler maintenance.

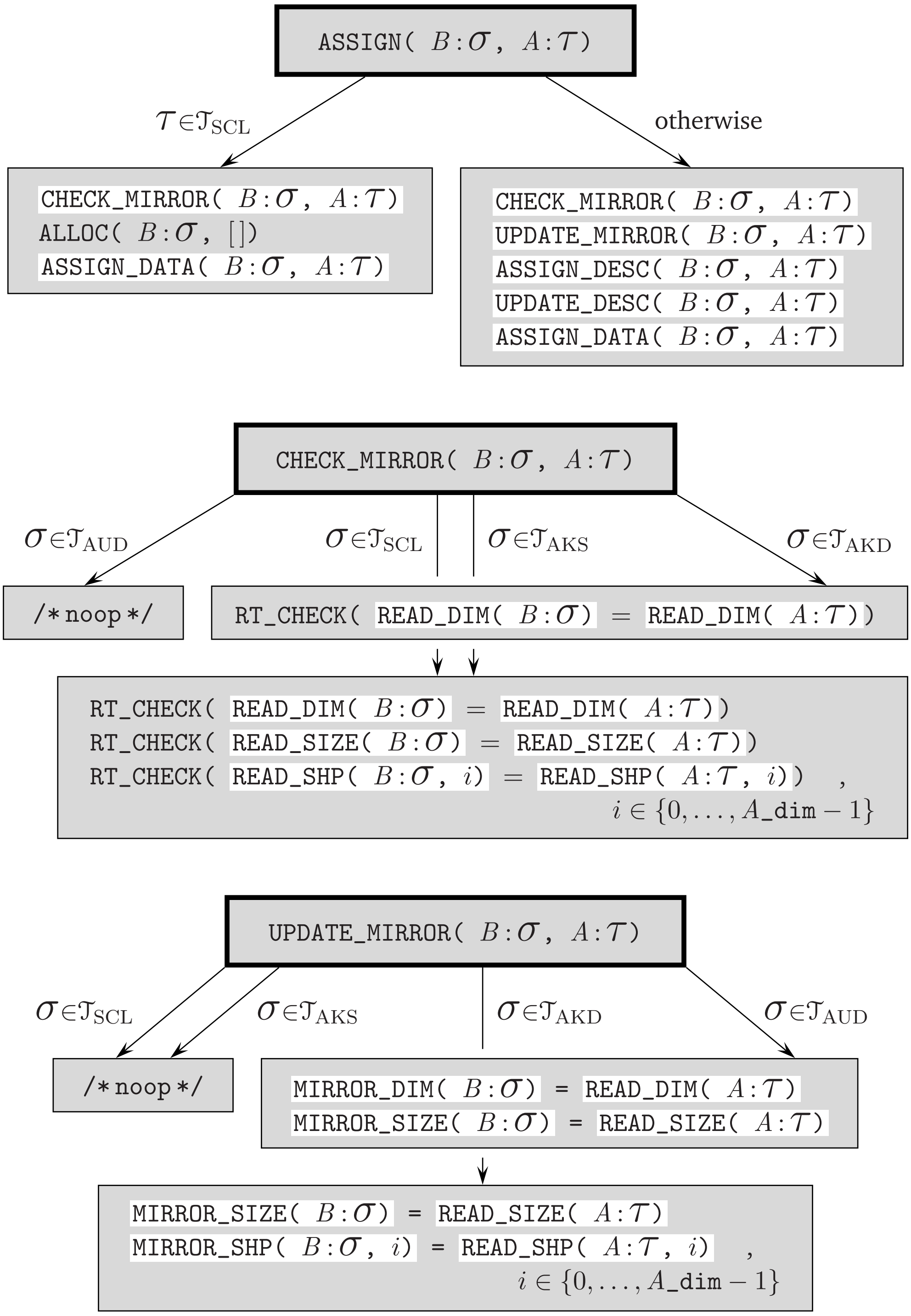


Figure 4.10: Implementation of the intermediate code macro `ASSIGN` *(first part).*

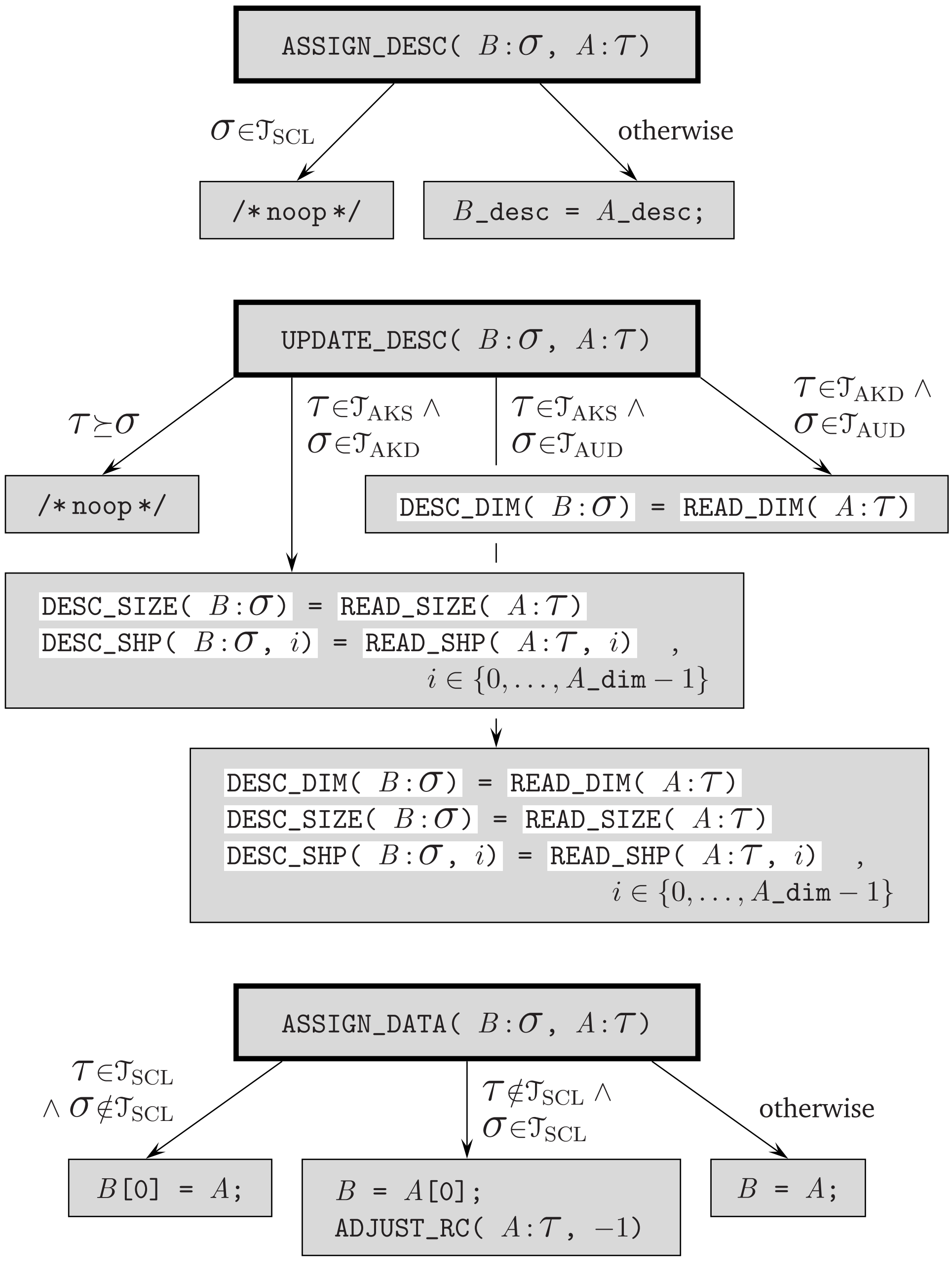


Figure 4.11: Implementation of the intermediate code macro `ASSIGN` *(second part).*

4.4 About the Backend Implementation

The compilation scheme described in this chapter has been fully integrated into the existing SAC compiler.[5] The compiler is written in ANSI C, consists of approximately 270.000 lines of source code, and has been under constant development since 1995.

In order to integrate the new compiler backend, the following tasks have been completed:

- A new compiler phase has been implemented which creates the wrapper functions needed to resolve function overloading (approximately 2.500 lines of source code).
- The new code generator has been implemented (approximately 40.000 lines of source code).
- The high-level code optimizations already integrated into the compiler have been initially invented in a non-generic setting where all arrays were supposed to have statically known shapes. Some of these (e. g., variable propagation, index vector elimination) have been generalized in order to be applicable to generic programs as well.
- The existing private heap manager for SAC programs has been adapted to the new array representations used.
- The existing runtime library needed for multi-threaded program execution has been modified to meet the requirements of the new backend.
- The existing interfaces between the languages SAC and C ("calling C functions from SAC programs" and "calling SAC functions from C programs") have been adapted to the new array representations used.
- The new compiler release has been tested on a large suite of SAC programs for several different combinations of hardware platforms and operating systems (UltraSPARC/Solaris, Intel x86/Linux, DEC Alpha/AIX, Mac PowerPC/Mac OS X, Intel x86/NetBSD).

However, owing to time limitations, some parts of the compiler are still incomplete or need further improvements:

[5] The source code of the SAC compiler as well as binary distributions for several different hardware platforms are available via WWW on the homepage of the SAC project: `http://www.sac-home.org/` .

- The runtime performance of the code generated by the compiler backend critically depends on the quality of the type inference system — the more specific the inferred shapes, the better the generated code. However, the recent revision of the type inference system delivers suboptimal array types in certain situations. Take as an example the following SAC code fragment:

```
sv = [10,10];
B = genarray( sv, 0);
```
.

  Having inferred that sv has type int$[2]$ does not suffice to deduce that B has type int$[10, 10]$. Instead, the value of sv must be taken into account as well. For this purpose it would be useful to extend the hierarchy of array types by a fifth type category: arrays with known shape and known value.

- For the time being, the compiler uses a rather naïve strategy for function specialization only. Traversing SAC programs from outermost to innermost, the compiler specializes all functions with respect to the types involved until the number of instances exceeds a predefined bound. Unfortunately, this may lead to very unfavorable results. For instance, applying the function Det (see Figure 2.8) to an argument of shape $[10, 10]$ would trigger the compiler to build specializations for argument shapes $[9, 9]$, $[8, 8]$, $[7, 7]$, and so on. But in order to get utmost runtime performance, it would be much more promising to build instances with small shapes first (i. e., $[3, 3]$, $[4, 4]$, $[5, 5]$, etc.) since these instances are predominantly used during the computation and offer much better potential for code optimizations (e. g., with-loop unrolling). Therefore, the SAC compiler provides means to control the function specialization explicitly through program annotations. Since this contradicts the ideals of high-level declarative programming, an optimized implicit control strategy for function specialization is most desirable.

- Some of the high-level code optimizations are not yet implemented for arrays of unknown shape. In principle, many of these optimizations could be generalized in order to be applicable to generic programs as well. However, especially for rather complex optimizations like with-loop folding this requires further research and programming effort.

- Careful examination of the output of the code generator reveal the demand for some low-level optimizations. Consider as an example explicit accesses to shape components of an array, i. e., nestings of the primitive operations shape and sel:

```
a = sel( [i], shape( A));
```
.

During the high-level code optimization phase such nestings are transformed into the following code:

```
sv = shape( A);
a = sv{i};
```
.

Here, the shape vector `[i]` has been replaced by a scalar, but if the shape of `A` is statically unknown, their is no way to eliminate the array `sv` as well at this stage of compilation. Nevertheless, `sv` is superfluous since the demanded shape component could be read directly from the array representation by means of a single intermediate code instruction:

```
a = READ_SHP( A, i);
```
.

This optimization is particularly important for wrapper functions which often contain lots of these shape accesses (e.g. Figure 4.2, lines 18 ff.).

Chapter 5

Performance Evaluation

This chapter evaluates the runtime behavior of the code generated by the new compilation scheme described in the preceding chapter.

Note here, that these performance measurements are fairly preliminary since they are based on rather simple SAC programs only. However, the temporary shortcomings of the current compiler implementation (as mentioned in Section 4.4) render the use of more complex examples impossible.

Nevertheless, all measurements have been performed on two different hardware platforms:

- A Sun UltraSPARC running Solaris-8, using the Sun C compiler (`cc` v5.2) as backend compiler. In the following, this platform will be denoted as UltraSPARC/Solaris.
- An Intel PentiumPro running Linux (Kernel v2.4.20), using the GNU C compiler (`gcc` v3.3). This platform will be denoted as i686/Linux.

The performance evaluation is intended to validate three major claims of this thesis:

- The design decision to use four different array representations during code generation leads to substantial runtime improvements. The benefit achieved by using individual representations for each type category easily outweighs the converting costs.
- Static shape inference and function specialization matter. The more specific the inferred shapes of a SAC program the better is the runtime performance of the generated C code. Hence, using the whole hierarchy of array types is essential for obtaining utmost runtime performance.

- The compilation scheme developed here generates code with a competitive runtime performance even if static shape inference fails for some parts of the source program.

Section 5.1 measures the costs of converting arrays from one representation into another. Subsequently, Section 5.2 analyses the runtime behavior of each primitive array operation depending on the type category of the argument array and depending on the array representations used for the generated code. Finally, Section 5.3 evaluates the function `Det` which has been used as running example in the preceding sections. Again, runtimes are measured depending on the array representations used for the generated code. Moreover, the impact of function specialization is investigated.

5.1 Conversion of Array Representations

An array has to be converted from one representation into another, only if an assignment of the form

$$\boxed{\texttt{B}:\sigma\ \texttt{=}\ \texttt{A}:\tau\texttt{;}}$$

occurs in the SAC code. The representations used for source and target array depend on the type categories ($\mathcal{T}_{\mathrm{SCL}}$, $\mathcal{T}_{\mathrm{AKS}}$, $\mathcal{T}_{\mathrm{AKD}}$, $\mathcal{T}_{\mathrm{AUD}}$) of which the types τ and σ are elements. Hence, with respect to type categories 16 different combinations are possible. However, some of these combinations are illegal. If one of the arrays is a scalar, the other array should be a (potential) scalar as well — situations like $\tau \in \mathcal{T}_{\mathrm{AKS}}$ and $\sigma \in \mathcal{T}_{\mathrm{SCL}}$, which will definitely cause a type error at runtime, are sorted out by the type inference system already. Thus, 12 legal combinations remain — among these 3 combinations on scalars and 9 combinations on non-scalars.

The runtime demand of such an assignment depending on the type category of the source (`A`) and the target array (`B`) is depicted in Figure 5.1 (UltraSPARC/ Solaris) and Figure 5.2 (i686/Linux). Note here, that the measurements are performed for a variable `A` which represents an integer array of shape $[10, 10, 10]$ in all non-scalar cases.[1]

[1]The conversion costs depend solely on the size of the array descriptors involved (see definition of the intermediate code macro `ASSIGN` in Figure 4.10). Hence, the costs are proportional to the dimension of `A`.

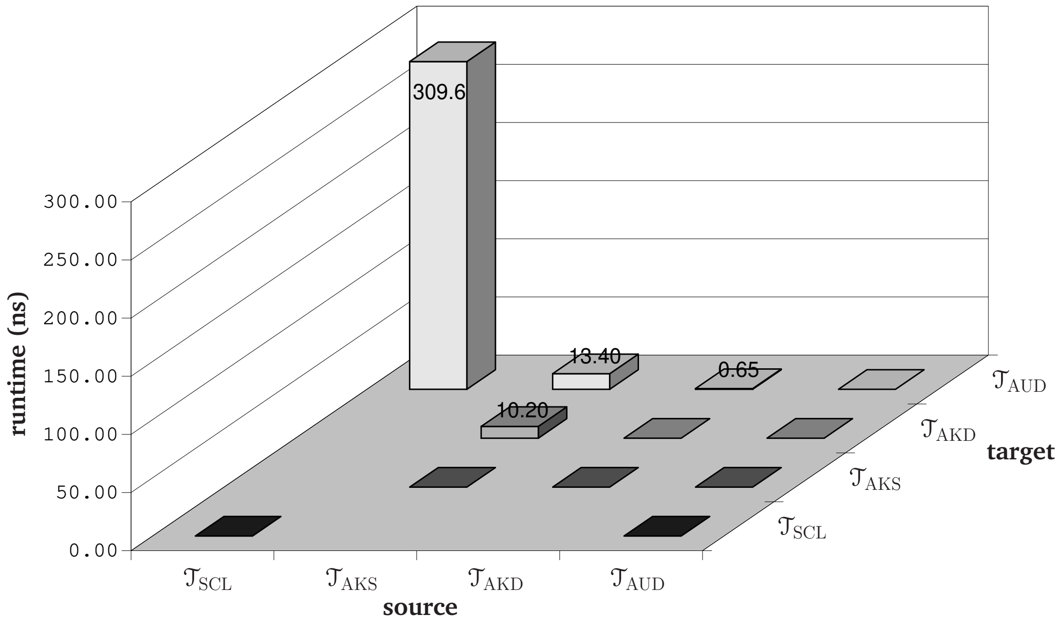


Figure 5.1: Time demand of converting the array representations on UltraSPARC/Solaris.

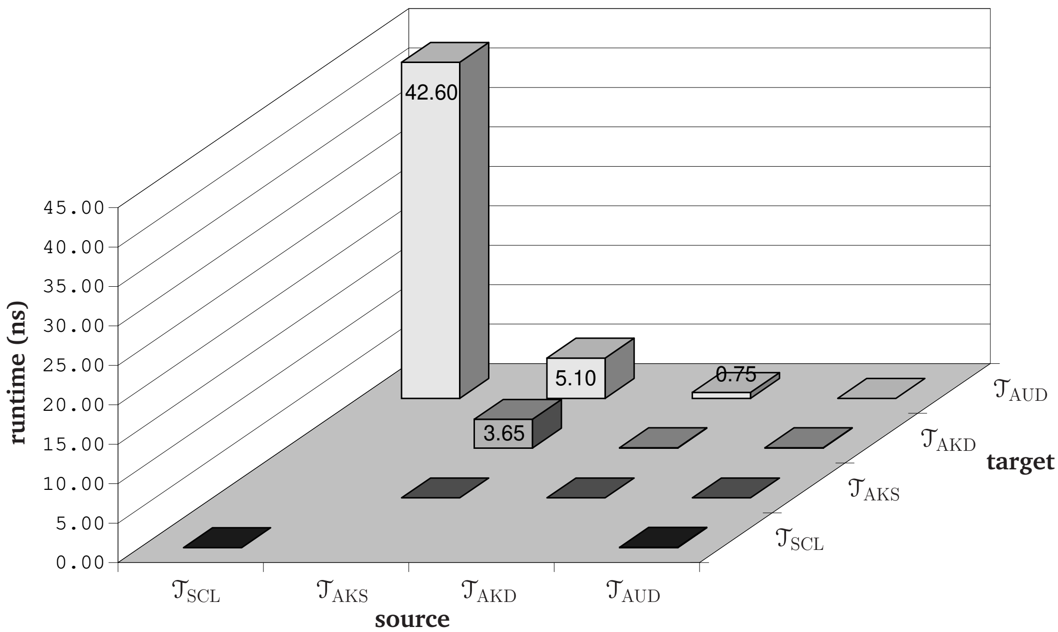


Figure 5.2: Time demand of converting the array representations on i686/Linux.

Both figures indicate that the conversion costs are significant in four cases only: $\mathcal{T}_{\mathrm{AUD}} = \mathcal{T}_{\mathrm{SCL}}$, $\mathcal{T}_{\mathrm{AUD}} = \mathcal{T}_{\mathrm{AKS}}$, $\mathcal{T}_{\mathrm{AUD}} = \mathcal{T}_{\mathrm{AKD}}$, and $\mathcal{T}_{\mathrm{AKD}} = \mathcal{T}_{\mathrm{AKS}}$. The first case mentioned is by far the most expensive one since it requires new descriptor memory to be allocated. Nevertheless, the runtime results given in the following sections will show that the benefits achieved by individual array representations easily outweigh this overhead.

5.2 Primitive Array Operations

The next runtime measurements are based on the primitive array operations provided by SAC. Again, let the variable `A` represent an integer array of shape $[10, 10, 10]$. The code segments[2] to be evaluated are defined as follows:

- Computing the dimension of the array `A`:

  ```
  B = dim( A:T);
  ```
  .

- Computing the shape of the array `A`:

  ```
  B = shape( A:T);
  ```
  .

- Performing a `reshape` operation on the array `A`:

  ```
  sv = [100,10];
  B = reshape( sv, A:T);
  ```
  .

  Note that `reshape` is applied to the last reference of `A`, hence, the operation is performed as destructive update. Otherwise, the runtime demand for copying the array would dominate the impact of shape inference and individual array representations.

- Selecting an element of the array `A`:

  ```
  iv = [3,...,2];   (over 3,...,2: dim( A:T))
  B = sel( iv, A:T);
  ```
  .

[2]Runtime costs of the surrounding program context, such as memory allocation for the array `A`, have been eliminated.

To minimize the impact of the internal hardware cache(s), the measurement is repeated several times with random values for the index vector `iv`.

- Modifying an element of the array `A`:

$$\begin{array}{|l|}\hline \texttt{iv = [}\overbrace{\texttt{4,...,7}}^{\texttt{dim(A:}\mathcal{T}\texttt{)}}\texttt{];} \\ \texttt{B = modarray(A:}\mathcal{T}\texttt{, iv, 16);} \\ \hline \end{array} \quad .$$

  Again, the value of `iv` is chosen by random, and `modarray` is applied to the last reference of `A` in order to allow a destructive update.

The preceding section has inferred the runtime overhead which is introduced by defining individual array representations for each type category. Now, the examples given above are well suited to determine the benefit of this design decision. In order to do so, a runtime flag has been incorporated into the SAC compiler which allows it to restrict the set of array representations to be used. The four representations can be ordered with respect to generality. The SCL- and the AKS-representation are applicable solely for the type categories $\mathcal{T}_{\mathrm{SCL}}$ and $\mathcal{T}_{\mathrm{AKS}}$ respectively. The AKD-representation can be used for $\mathcal{T}_{\mathrm{AKS}}$ as well as $\mathcal{T}_{\mathrm{AKD}}$, whereas the AUD-representation is applicable for all categories. Hence, the following four scenarios are conceivable:

- The compiler uses all four array representations. This is the default scenario.
- The compiler uses the SCL-, AKD-, and AUD-representation only. Arrays of type $\in \mathcal{T}_{\mathrm{AKS}}$ are implemented using the AKD-representation.
- The compiler uses the SCL- and the AUD-representation only. Arrays of type $\in \mathcal{T}_{\mathrm{AKS}} \cup \mathcal{T}_{\mathrm{AKD}}$ are implemented using the AUD-representation.
- The compiler uses the AUD-representation only.

It is readily apparent that the last scenario causes dramatic performance losses (multiple orders of magnitude!). Most of the optimization techniques implemented in the SAC compiler try to replace arrays by sets of scalars wherever possible. Implementing all scalars as boxed arrays would surely pervert these measures. Hence, only the first three scenarios will be looked at. These are characterized by the minimal (i. e., most specific) representation available for non-scalar arrays. That is AKS for the first, AKD for the second, and AUD for the third scenario.

Another criterion which potentially affects the runtime efficiency of the generated code is the type which has been inferred for the argument `A`. The more specific the inferred shape, the better the potential for high-level code optimizations during compilation. Therefore, the measurements are performed with three different values for $\mathcal{T}$:

- $\mathcal{T} \equiv \texttt{int}[10, 10, 10] \in \mathfrak{T}_{\mathrm{AKS}} \quad ,$
- $\mathcal{T} \equiv \texttt{int}[\bullet,\bullet,\bullet] \in \mathfrak{T}_{\mathrm{AKD}} \quad ,$
- $\mathcal{T} \equiv \texttt{int}[\texttt{+}] \in \mathfrak{T}_{\mathrm{AUD}} \quad .$

The measurement results are depicted in the figures on the following pages. Each figure consists of three groups of bars, where the groups correspond to the type category of the argument `A`, and the bars within each group correspond to the minimal array representation used for the generated code.

In general, both criteria have a significant impact on the runtime efficiency. Static shape inference reduces the time demand up to a factor of 3 on UltraSPARC/Solaris and up to a factor of 1.5 on i686/Linux. For real world applications this effect will be considerably higher since the most promising code optimizations (e. g., common subexpression elimination, constant folding, `with`-loop folding) are relevant for sequences of multiple array operations only. The impact of the tailor-made array representations is in most cases much smaller. Nevertheless, the time demand of each array operation — apart from the rather trivial operation `dim` — is decreased by at least $5\,\%$ which easily outweighs the runtime overhead of a single conversion operation.

The evaluation results in more detail: Figures 5.3 and 5.4 depict the results for the operation `dim` on UltraSPARC/Solaris and i686/Linux. Having inferred the dimension of `A`, the compiler is able to replace the whole operation by the constant value 3. Hence, the time demand is negligible in these cases.

The time demand of the operation `shape` is given in Figures 5.5 and 5.6. This operation returns a vector which is filled with the shape components of `A`. If `A` is implemented using an AUD-representation — i. e., the type category of `A` is $\mathfrak{T}_{\mathrm{AUD}}$ *or* the minimal array representation used is AUD — these shape components are not found in mirror variables but must be read from the descriptor. Since this requires costly accesses to the main memory, a substantial loss of runtime efficiency occurs.

Figures 5.7 and 5.8 show the runtimes of the operation `reshape`. Since `reshape` is applied to the last reference of `A`, the operation can more or less be implemented as a simple assignment (see compilation rule on page 67). However, the variations in time demand are much more dramatic than expected. This effect is most likely caused by systematic cache conflicts: Small modifications with respect to memory accesses could have a considerable impact on the efficiency of the internal hardware cache(s) which in turn leads to disproportionate slowdowns.

The operation `sel` is addressed in Figures 5.9 and 5.10. If the compiler succeeds in inferring at least the dimension of `A`, the index vector `iv` is replaced by a scalar during code optimizations (see *index vector elimination* on page 29) which reduces the runtimes by $40\,\%$ on UltraSPARC/Solaris and by $12\,\%$ on i686/Linux. In case $\mathcal{T} \in \mathfrak{T}_{\mathrm{AUD}}$ the length of `iv` is statically undecidable, hence, index vector elimination is not applicable.

Finally, Figures 5.11 and 5.12 depict the runtimes for the operation `modarray`. Again, index vector elimination is performed for $\mathcal{T} \in \mathfrak{T}_{\mathrm{AKS}} \cup \mathfrak{T}_{\mathrm{AKD}}$ which leads to a performance gain of up to $70\,\%$ and $20\,\%$ respectively.

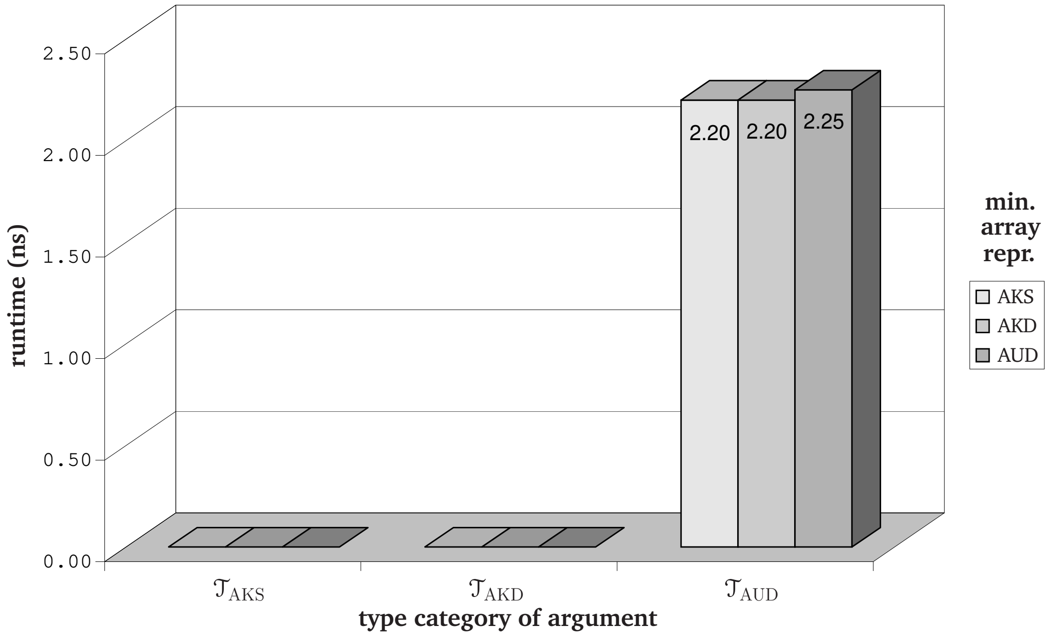


Figure 5.3: Time demand of the primitive operation `dim` *on UltraSPARC/Solaris.*

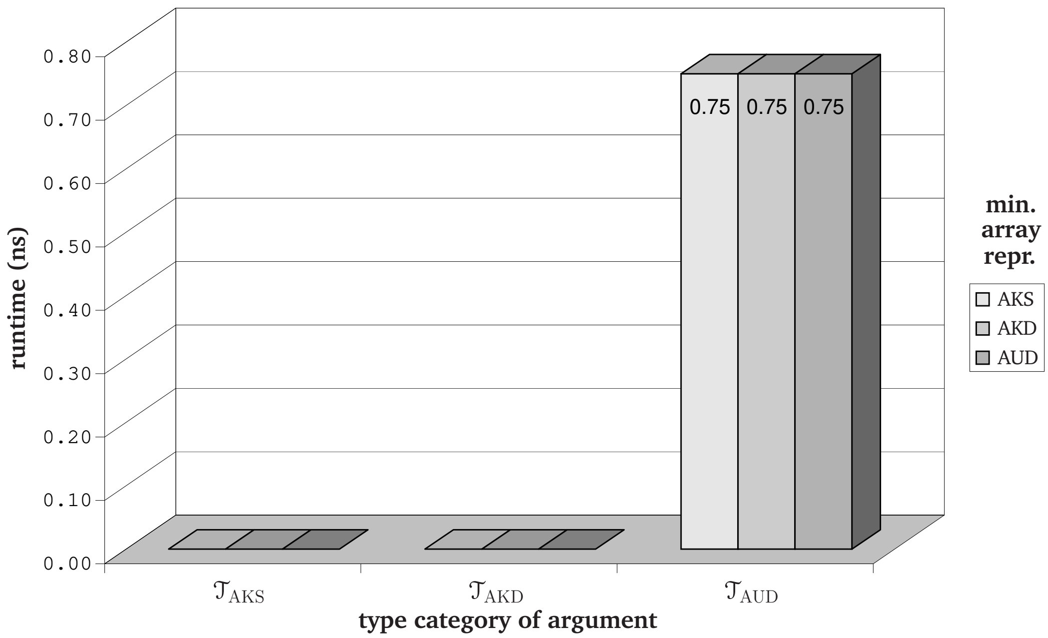


Figure 5.4: Time demand of the primitive operation `dim` *on i686/Linux.*

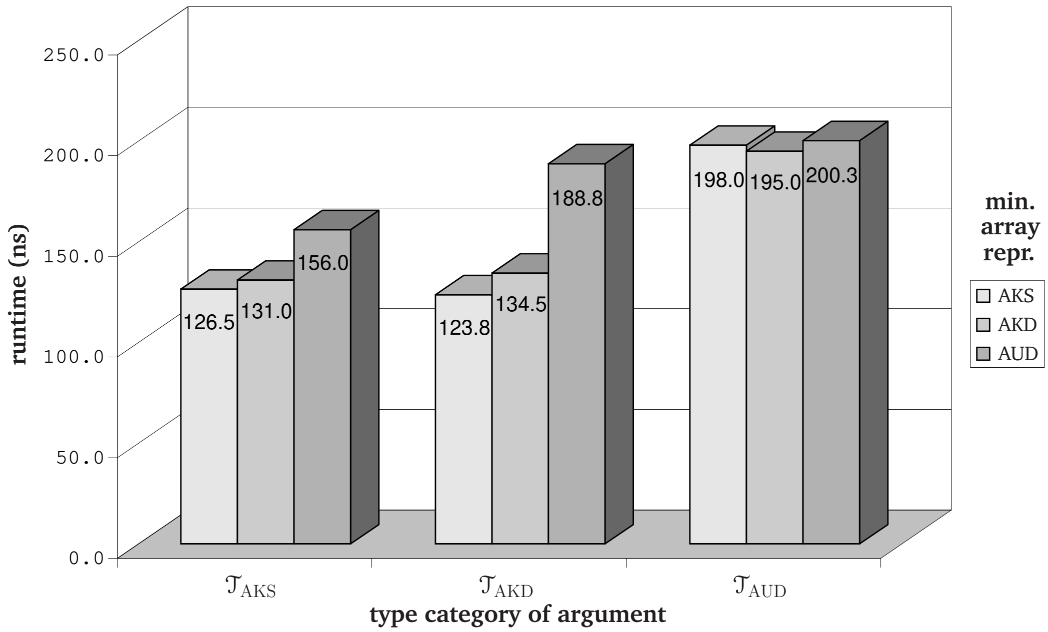


Figure 5.5: Time demand of the primitive operation `shape` *on UltraSPARC/Solaris.*

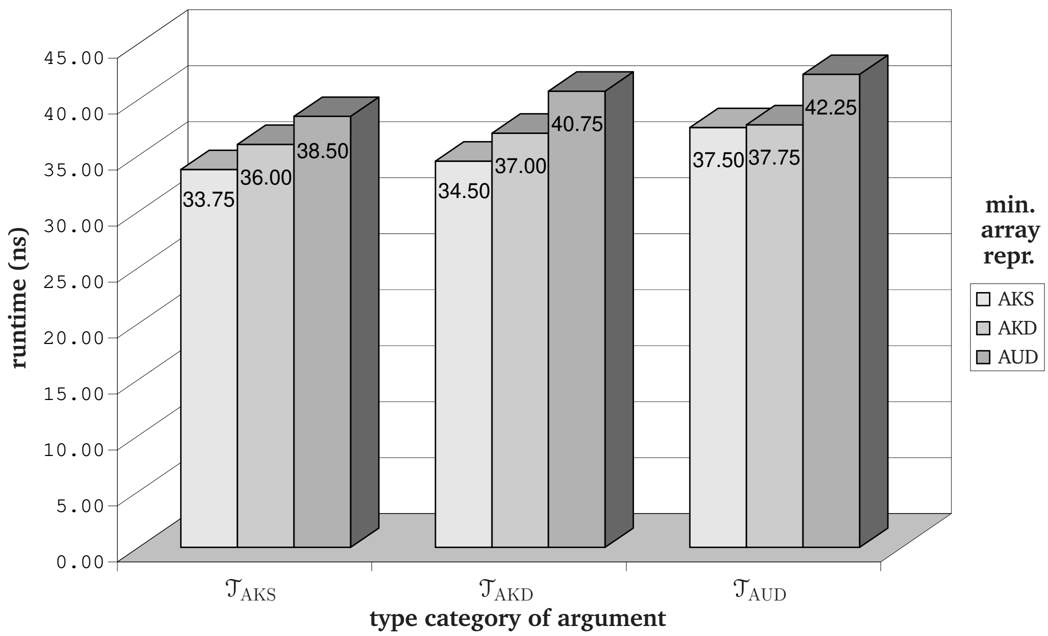


Figure 5.6: Time demand of the primitive operation `shape` *on i686/Linux.*

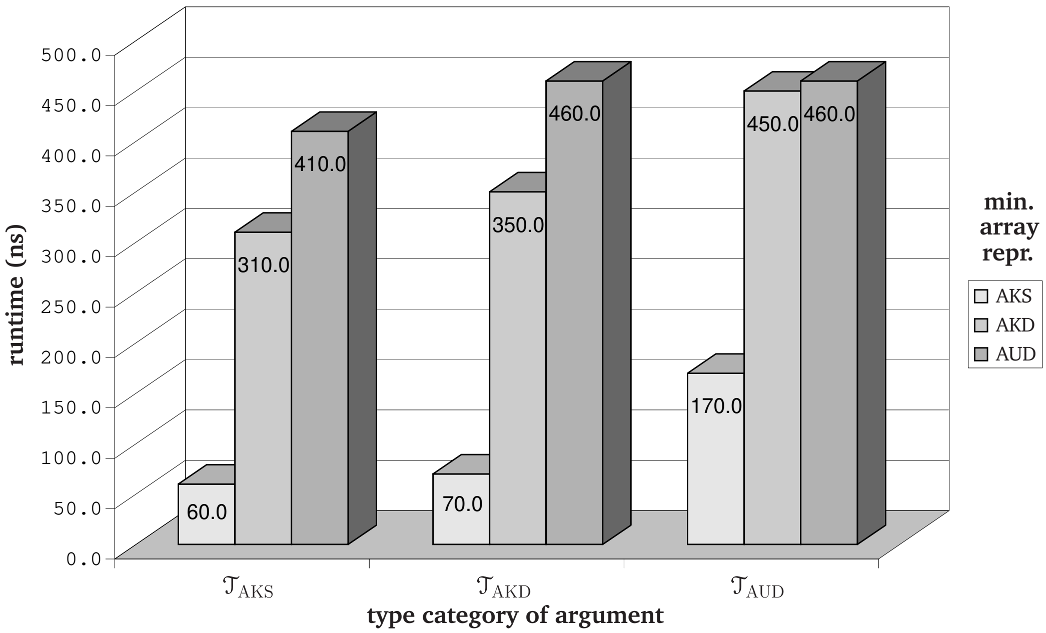


Figure 5.7: Time demand of the primitive operation `reshape` on UltraSPARC/Solaris.

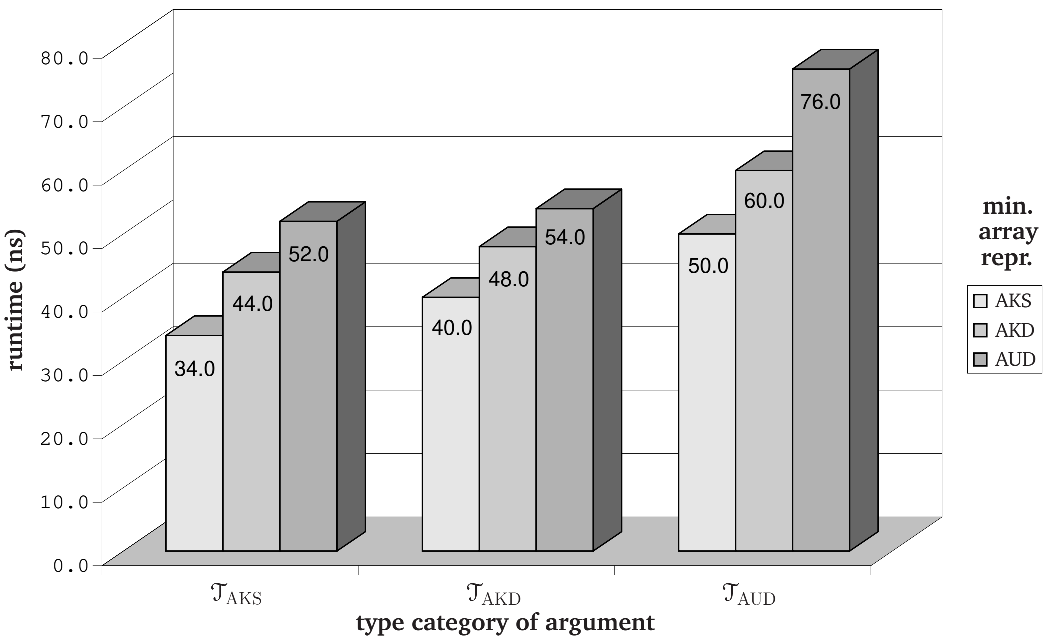


Figure 5.8: Time demand of the primitive operation `reshape` on i686/Linux.

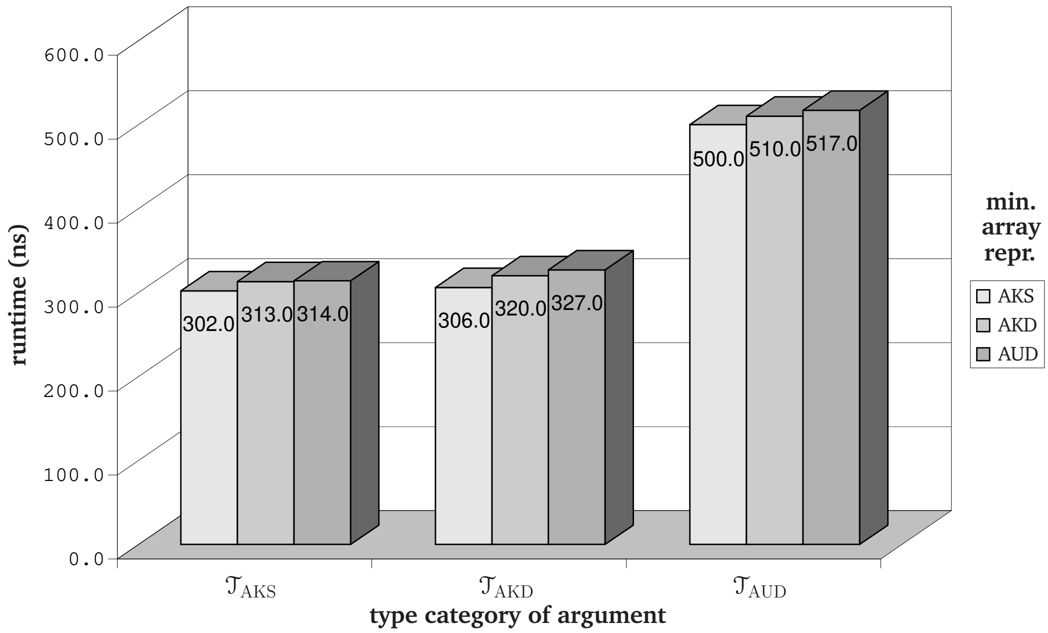


Figure 5.9: Time demand of the primitive operation `sel` *on UltraSPARC/Solaris.*

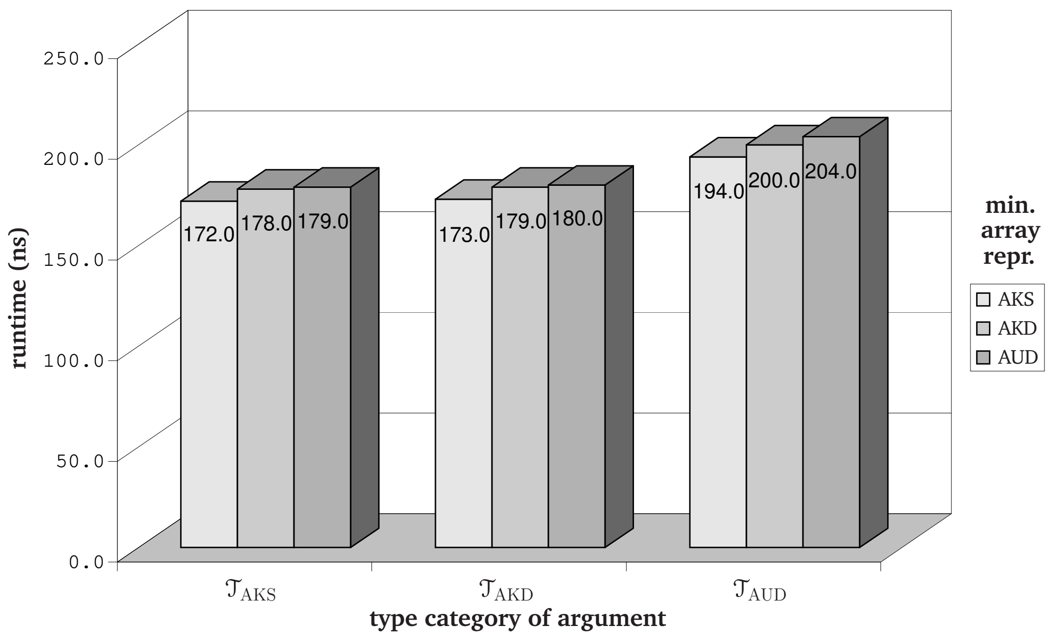


Figure 5.10: Time demand of the primitive operation `sel` *on i686/Linux.*

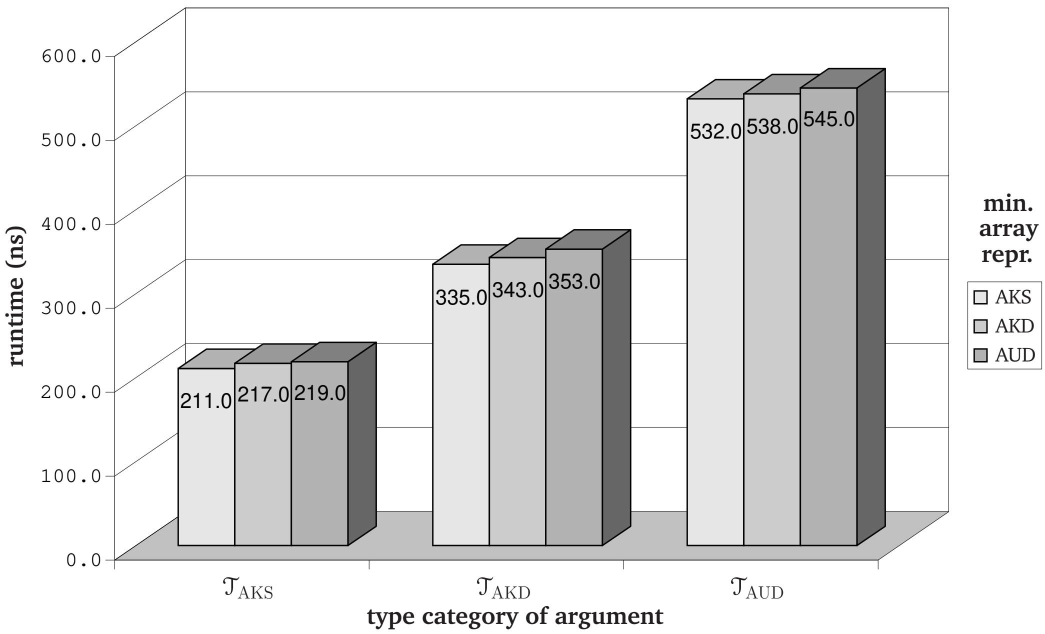


Figure 5.11: Time demand of the primitive operation `modarray` *on UltraSPARC/Solaris.*

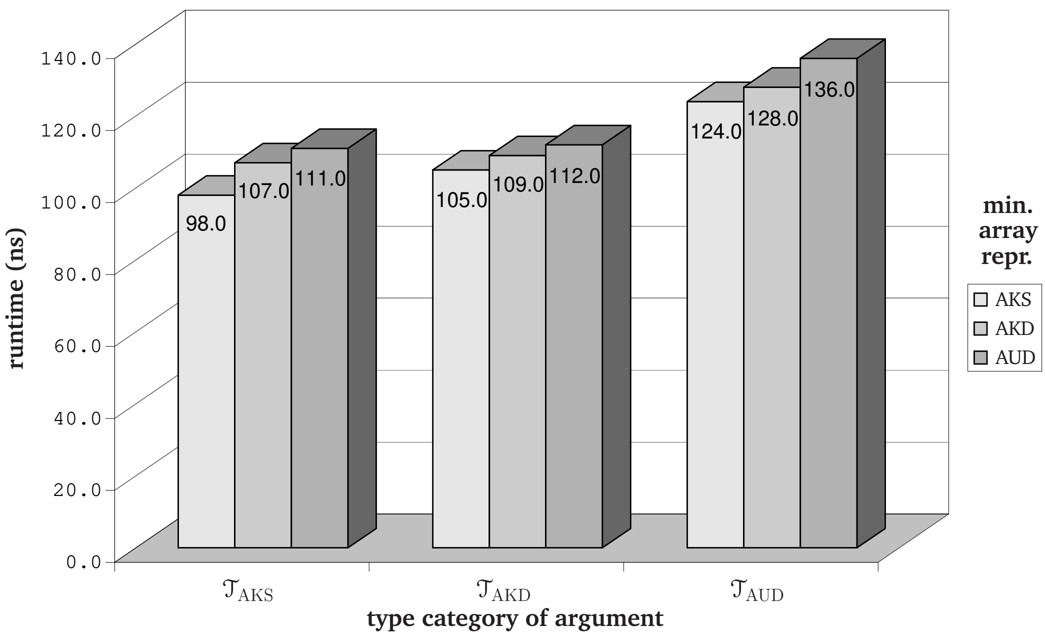


Figure 5.12: Time demand of the primitive operation `modarray` *on i686/Linux.*

5.3 A Case Study: Determinant

Having examined the runtime behavior of the code generated for isolated SAC assignments, this chapter evaluates a complete real-world application — a program which computes the determinant of a 10×10 array:

```
int main()
{
  int[10,10] a;
  int det;

  a = ...;
  det = Det( a);

  return( det);
}
```
,

where the function Det is defined as depicted in Figure 2.8 on page 20.

The performance evaluation is intended to investigate the overall impact of using tailor-made representations for the different type categories. The question is, whether the aggregated performance gains and losses really lead to significant runtime improvements or not. Therefore, the measurements are again performed with three different minimal array representations.

Moreover, the example given above is well-suited to demonstrate that function specialization is essential to create code with utmost runtime efficiency. For this purpose, during compilation five different strategies for function specialization are used:

- The compiler builds no specializations at all, i. e., only the original two instances of the function Det are available. Whenever Det is applied to an argument whose shape is not $[2, 2]$, the generic instance is used. In the following, this strategy will be denoted as “generic”.
- The compiler builds a single specialization of the function Det for argument shape $[3, 3]$. In the following, this strategy will be called “generic 3”.
- The compiler builds specializations for argument shapes $[3, 3]$ and $[4, 4]$. This strategy will be called “generic 4”.
- The compiler builds specializations for argument shapes $[3, 3]$, $[4, 4]$, and $[5, 5]$. This strategy will be denoted as “generic 5”.

- The compiler builds instances for all needed argument shapes $[10, 10]$, $\dots$, $[3, 3]$. As a consequence, all array variables have statically known shapes and function overloading can be resolved statically. This strategy will be called “specific”.

The results of the runtime measurements are shown in Figures 5.13 and 5.14. The five groups of bars relate to the specialization strategy applied, whereas the color of each bar indicates the minimal array representation used. The additional marks on the vertical axis at $2.11\mathrm{s}$ and $0.19\mathrm{s}$ respectively denote the time demand of an equivalent C implementation of the `Det` program.

The runtime figures show that the tailor-made array representations have a significant impact on the execution times of the generated code since they decreases the execution times by $14{-}20\,\%$ ($8{-}28\,\%$) on UltraSPARC/Solaris (on i686/Linux).

Furthermore, the measurements demonstrate that specializing functions is indeed crucial for getting best possible runtime performance. The more specializations are built by the compiler, the lower is the time demand of the generated code. The generic version without any specializations is about a factor of 5.0 (8.0) slower than the fully specialized version. Building one or two specialized instances of the function `Det` reduces the slowdown to a factor of 2.3 (3.0) or 1.4 (1.8) respectively.

However, it is also indicated that by means of the new compilation scheme even generic functions can be compiled into code with an acceptable runtime performance. Note, that it suffices to build a single (two) specialization(s) of the function `Det` to get approximately the same execution time as the C implementation. If the compiler adds additional specializations, the SAC implementation is significantly faster than the C implementation.

It should be noted that the algorithm for determinant computation used here has a factorial complexity. In practice, other and much more efficient algorithms like matrix orthogonalization are predominantly used instead [Stoe99, PTVF92]. Nevertheless, the Laplace expansion offers the opportunity to investigate the runtime behavior of a function, whose argument shape is changing with each recursive call, in a very clean setting. In contrast to other algorithms of this kind — the most prominent example being multigrid relaxation[3] — it is easy to describe and not burdened with any distracting frills.

[3] Multigrid relaxation is used to approximate solutions for discrete Poisson equations. Each iteration step consists of relaxation steps and smoothing steps which are recursively embedded within operations to coarsen and refine grid granularities [HT82, Bran84, Hack85].

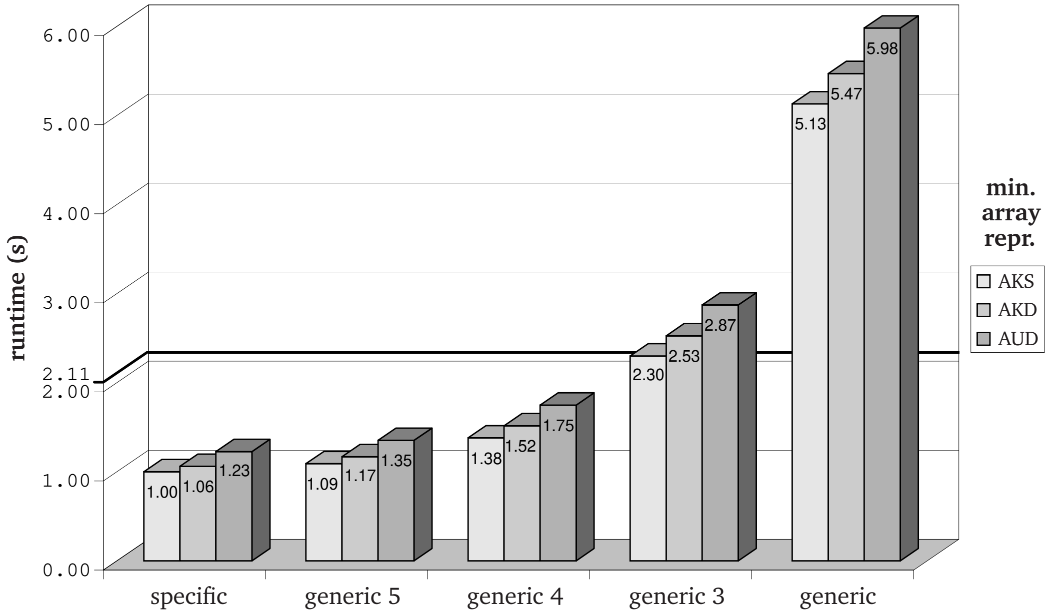


Figure 5.13: Time demand of computing the determinant of a 10×10 *array on UltraSPARC/Solaris.*

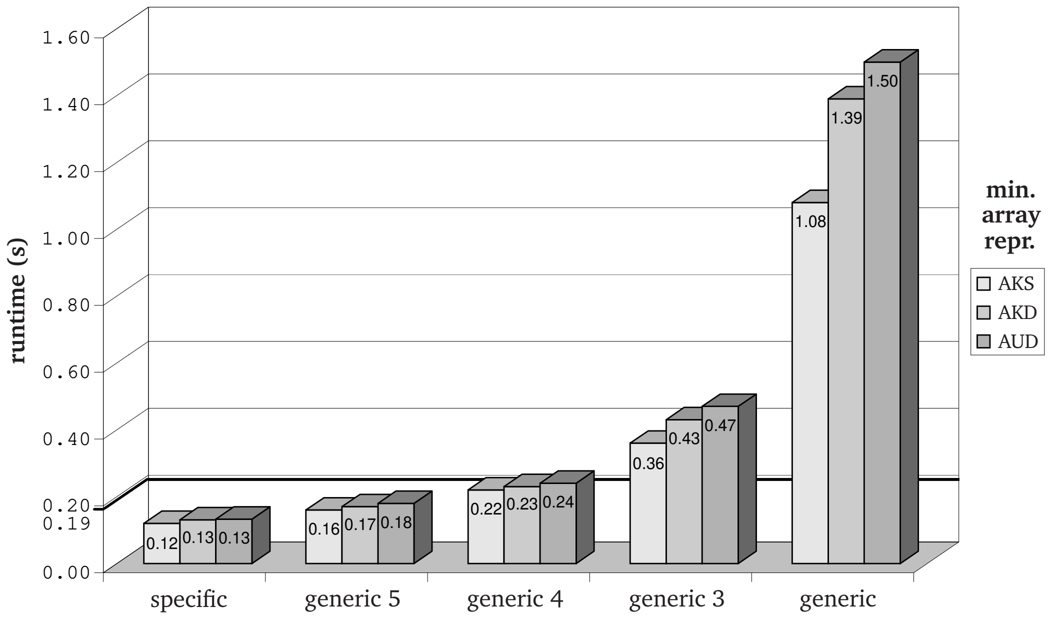


Figure 5.14: Time demand of computing the determinant of a 10×10 *array on i686/Linux.*

Even so, the same performance evaluation has been done for a shape-invariant SAC implementation of multigrid relaxation as well. Unfortunately, owing to the mentioned limitations of the current compiler implementation, the obtained results are unsatisfactory and not worth to be depicted here. However, since former runtime measurements on multigrid in a non-generic setting have achieved excellent results [Grel02], it can be assumed that an improved revision of the type inference system will lead to runtime figures similar to those for the Laplace expansion.

Chapter 6

Conclusion

One of the key features of array processing languages is the support for shape-invariant programming, i. e., all array operations can be defined in a generic way that allows arguments to have arbitrary dimension and size. The advantages of this programming technique are manifold. First, it eases program development since complicated loop nestings as well as the explicit and error-prone specification of array indices can often be avoided. Moreover, it improves the generality of programs and thus simplifies reuse and maintenance of existing programs.

However, when trying to compile such generic array operations into efficiently executable code, static knowledge of exact array shapes is essential. Therefore, modern compilers try to infer the shapes of all arrays used in a program.

Unfortunately, static shape inference is generally undecidable, e. g., if input data have unknown shapes, or if a recursive function is applied to an argument whose shape is changing with each recursive call. Hence, recent compilers either rule out all programs for which shape inference fails, or they perform no shape inference at all. In the first case the expressive power of the language is significantly restricted, in the latter case the generated code has a poor runtime performance.

This thesis develops a new compilation scheme for the language SAC which combines these two approaches in order to avoid their individual shortcomings. It generates shape-specific code whenever exact shapes can be statically inferred and generates more generic code, otherwise. The basic idea is to make use of a hierarchy of array types with different levels of shape information — the most specific array types specify an exact shape, whereas more general types prescribe an exact dimension only or contain no shape information at all — and to translate these types into a corresponding hierarchy of array representations.

Since the number of different array shapes is infinite, this hierarchy of array types is unbounded. Nevertheless, it can be classified into four categories of types: scalar arrays, non-scalar arrays with known shape, non-scalars with known dimension but unknown extent, and non-scalars with unknown dimension.

Compiling shape-invariant SAC programs into efficiently executable code is done as follows: At first, the compiler infers the types of all local variables. In order to achieve utmost potential for code optimizations, the compiler tries to infer types as shape-specific as possible. Therefore, generically defined functions are specialized with respect to the required argument shapes.

Subsequently, the compiler has to resolve function overloading. In SAC, functions can be overloaded not only with respect to base types but also with respect to shapes, i. e., SAC programs might contain shape-specific as well as generic instances of a single function. If static shape inference fails, it may not be statically decidable which instance of a function has to be used for a given application. Hence, the compiler must generate additional code which performs appropriate type checks and chooses the matching instance at runtime. In principle, for each function application, a tailor-made code fragment is needed which resolves the overloading. However, the approach suggested in this thesis achieves this by means of an elegant high-level code transformation. For each overloaded function a generic wrapper, which is also written in SAC, is generated that resolves the overloading for the most general case only. Afterwards, the code of this wrapper function is individually adapted to each function application by means of the usual code optimizations already integrated into the compiler.

In the final compilation step the optimized SAC code is transformed into semantically equivalent C code. An important issue in this context is to find an appropriate C representation for SAC arrays. In order to get code with an acceptable runtime performance, it is essential that arrays which are identified as scalars are represented in C by scalars as well. Other SAC arrays could be uniquely implemented in C by means of a descriptor and a data vector. However, performance measurements indicate that the runtime demand of the generated code can be further reduced by using not only two but four different array representations — one for each type category.

Unfortunately, using multiple array representations in a generic setting has a dramatic impact on the complexity of the code generator. Primitive array operations or generically defined functions are applicable to arrays of any type category. Therefore, it is often necessary to convert arrays from one representation into another. Thus, the array representations are defined in a way that minimizes these conversion costs. Moreover, the generated code must be individually adapted to the array representations involved. For a primitive function which requires three arguments with four different representations each, the code generator must take up to $4^3 = 64$ different cases into account. This complexity problem is solved by means of a sophisticated transformation scheme which uses multiple layers of intermediate code macros.

In order to demonstrate the effectiveness of the compilation scheme, several runtime measurements are performed. The measurements confirm that the use of four instead of two different array representations has a significant impact on the runtime efficiency of the generated code. The obtained performance gain easily outweighs the costs of converting arrays from one representation into another. On average, the overall runtime demand is reduced by at least $10\,\%$.

However, the impact of static shape inference and function specialization is much stronger. Even for isolated primitive array operations it reduces the time demand by up to a factor of 3. For real world applications, like determinant computation or relaxation, the effect is considerably higher since sequences of multiple array operations offer higher potential for code optimizations.

The promising evaluation results notwithstanding, there are a few problems left that still need to be addressed. For the time being, the SAC compiler uses a suboptimal type inference algorithm as well as a rather naïve strategy for function specialization. As a consequence, the type inference system delivers unfavorable results in some situations. In order to exploit the full potential of the new compiler backend, it is necessary to eliminate these shortcomings.

Furthermore, the high-level code optimizations integrated into the compiler have been initially invented in a non-generic setting where all arrays were supposed to have statically known shapes. In principle, many of these optimizations could be generalized in order to be applicable to generic programs as well. However, owing to time limitations, especially rather complex optimizations like `with`-loop folding are sofar only implemented for arrays of known shape.

Bibliography

[ABM$^+$92] J. C. Adams, W. S. Brainerd, J. T. Martin, et al.: *Fortran-90 Handbook: Complete ANSI/ISO Reference*. McGraw-Hill, 1st edition, 1992. ISBN 0-07-000406-4.

[AK02] R. Allen, K. Kennedy: *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 1st edition, 2002. ISBN 1-55860-286-0.

[ANSI78] American National Standards Institute: *ANSI Fortran X3.9–1978*. Technical Report ANSI X3.9-1978, American National Standards Institute, 1978.

[Appe98] A. W. Appel: *Modern Compiler Implementation in C*. Cambridge University Press, 1st edition, 1998. ISBN 0-521-58390-X.

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers: Principles, Techniques, and Tools*. Series in Computer Science. Addison-Wesley, 1st edition, 1986. ISBN 0-201-10194-7.

[Bare84] H. P. Barendregt: *The Lambda Calculus: Its Syntax and Semantics*, Vol. 103 of: Studies in Logics and the Foundations of Mathematics. North-Holland, 2nd edition, 1984. ISBN 0-444-86748-1.

[BCOF91] A. P. W. Böhm, D. C. Cann, R. R. Oldehoeft, J. T. Feo: *Sisal Reference Manual Language Version 2.0*. CS 91-118, Colorado State University, Fort Collins, Colorado, USA, 1991.

[Bern93] R. Bernecky: *The Role of APL and J in High-Performance Computation*. In E. M. Anzalone (Ed.): Proceedings of the Array Processing Language Conference (APL '93), Toronto, Canada. Vol. 24(1) of: APL Quote Quad. ACM Press, 1993, pp. 17–32.

[Bern97] R. Bernecky: *APEX: The APL Parallel Executor*. Master's Thesis, University of Toronto, Canada, 1997.

[BGS94] D. F. Bacon, S. L. Graham, O. J. Sharp: *Compiler Transformations for High-Performance Computing*. ACM Computing Surveys, 26(4), pp. 345–420, 1994.

[Bird98] R. Bird: *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall, 2nd edition, 1998. ISBN 0-13-484346-0.

[Bran84] A. Brandt: *Multigrid Methods: 1984 Guide*. Technical Report, The Weizmann Institute of Science, Department of Applied Mathematics, Rehovot, Israel, 1984.

[Brow85] J. Brown: *Inside the APL2 Workspace*. ACM Quote Quad, 15, pp. 277–282, 1985.

[BSMM99] I. N. Bronstein, K. A. Semendjajew, G. Musiol, H. Mühlig: *Taschenbuch der Mathematik*. Harri Deutsch, 4th edition, 1999. ISBN 3-8171-2004-4.

[Budd88] T. Budd: *An APL Compiler*. Springer, 1st edition, 1988. ISBN 0-387-96643-9.

[Burk96] C. Burke: *J and APL*. Iverson Software Inc., Toronto, Canada, 1996.

[BW88] R. Bird, P. Wadler: *Introduction to Functional Programming*. Series in Computer Science. Prentice Hall, 1st edition, 1988. ISBN 0-13-484197-2.

[Cann89] D. C. Cann: *Compilation Techniques for High Performance Applicative Computation*. Technical Report CS-89-108, Lawrence Livermore National Laboratory, Livermore, California, USA, 1989.

[Cann92] D. C. Cann: *Retire Fortran? A Debate Rekindled*. Communications of the ACM, 35(8), pp. 81–89, 1992.

[Cann93] D. C. Cann: *The Optimizing Sisal Compiler (Version 12.0)*. Lawrence Livermore National Laboratory, Livermore, California, USA, 1993. Part of the Sisal distribution.

[CE95] D. C. Cann, P. Evripidou: *Advanced Array Optimizations for High Performance Functional Languages*. IEEE Transactions on Parallel and Distributed Systems, 6(3), pp. 229–239, 1995.

[CF58] H. B. Curry, R. Feys: *Combinatory Logic (Vol. 1)*. Studies in Logics and the Foundations of Mathematics. North-Holland, 1st edition, 1958. ISBN 0-7204-2208-6.

[Cohe81] J. Cohen: *Garbage Collection of Linked Data Structures*. ACM Computing Surveys, 13(3), pp. 341–367, 1981.

[CW85] L. Cardelli, P. Wegner: *On Understanding Types, Data Abstraction, and Polymorphism*. ACM Computing Surveys, 17(4), pp. 471–522, 1985.

[DO86] G. C. Driscoll, D. L. Orth: *Compiling APL: The Yorktown APL Translator*. IBM Journal of Research and Development, 30(6), pp. 583–593, 1986.

[FH88] A. J. Field, P. G. Harrison: *Functional Programming*. International Computer Science Series. Addison-Wesley, 1st edition, 1988. ISBN 0-201-19249-7.

[FMSD95] J. T. Feo, P. J. Miller, S. K. Skedzielewski, S. M. Denton: *Sisal-90 User's Guide*. Lawrence Livermore National Laboratory, Livermore, California, USA, 1995.

[FO95] S. M. Fitzgerald, R. R. Oldehoeft: *Update-in-Place Analysis for True Multidimensional Arrays*. In A. P. W. Böhm, J. T. Feo (Eds.): High Performance Functional Computing. 1995, pp. 105–118.

[GKS00] C. Grelck, D. Kreye, S.-B. Scholz: *On Code Generation for Multi-Generator WITH-Loops in SAC*. In P. Koopman, C. Clack (Eds.): Implementation of Functional Languages, 11th International Workshop (IFL '99), Lochem, The Netherlands, Selected Papers. Vol. 1868 of: Lecture Notes in Computer Science. Springer, 2000, pp. 77–94. ISBN 3-540-67864-6.

[Grel96] C. Grelck: *Integration eines Modul- und Klassen-Konzeptes in die funktionale Programmiersprache SAC — Single Assignment C*. Diploma Thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.

[Grel01] C. Grelck: *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C*. PhD Thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, Germany, 2001. ISBN 3-89722-719-3. Published by Logos.

[Grel02] C. Grelck: *Implementing the NAS Benchmark MG in SAC*. In: Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002), Fort Lauderdale, Florida, USA. IEEE Computer Society Press, 2002.

[Gron97] J. van Groningen: *The Implementation and Efficiency of Arrays in Clean 1.1*. In W. E. Kluge (Ed.): Implementation of Functional Languages, 8th International Workshop (IFL '96), Bad Godesberg, Germany, Selected Papers. Vol. 1268 of: Lecture Notes in Computer Science. Springer, 1997, pp. 105–124. ISBN 3-540-63237-9.

[GS95] C. Grelck, S.-B. Scholz: *Classes and Objects as Basis for I/O in SAC*. In T. Johnsson (Ed.): Proceedings of the 7th International Workshop on the Implementation of Functional Languages (IFL '95). Chalmers University of Technologie, Båstad, Sweden, 1995, pp. 30–44.

[GS00] C. Grelck, S.-B. Scholz: *HPF vs. SAC — A Case Study*. In A. Bode, T. Ludwig, W. Karl, R. Wismüller (Eds.): Euro-Par 2000, Parallel Processing, Proceedings of the 6th International Euro-Par Conference, Munich, Germany. Vol. 1900 of: Lecture Notes in Computer Science. Springer, 2000, pp. 620–624. ISBN 3-540-67956-1.

[GS03] C. Grelck, S.-B. Scholz: *Axis Control in SAC*. In R. Peña, T. Arts (Eds.): Implementation of Functional Languages, 14th International Workshop (IFL 2002), Madrid, Spain, Selected Papers. Lecture Notes in Computer Science. Springer, 2003.

[Hack85] W. Hackbusch: *Multi-Grid Methods and Applications*, Vol. 4 of: Series in Computational Mathematics. Springer, 1st edition, 1985. ISBN 3-540-12761-5.

[Hank94] C. Hankin: *Lambda Calculi: A Guide for Computer Scientists*. Graduate Texts in Computer Science. Oxford University Press, 1st edition, 1994. ISBN 0-19-853840-5.

[HB85] P. Hudak, A. Bloss: *The Aggregate Update Problem in Functional Programming Systems*. In: Proceedings of the 12th Symposium on Principles of Programming Languages (POPL '85), New Orleans, Louisiana, USA. ACM Press, 1985, pp. 300–314.

[HB93] M. Haines, A. P. W. Böhm: *Task Management, Virtual Shared Memory, and Multithreading in a Distributed Memory Implementation of Sisal*. In A. Bode, et al. (Eds.): Parallel Architectures and Languages Europe (PARLE '93), Munich, Germany. Vol. 694 of: Lecture Notes in Computer Science. Springer, 1993, pp. 12–23.

[HPFF97] High Performance Fortran Forum: *High Performance Fortran Language Specification V2.0*, 1997.

[HS86] J. R. Hindley, J. P. Seldin: *Introduction to Combinators and Lambda Calculus*, Vol. 1 of: London Mathematical Society Student Texts. Cambridge University Press, 1st edition, 1986. ISBN 0-521-31839-4.

[HT82] W. Hackbusch, U. Trottenberg (Eds.): *Multigrid Methods: Proceedings of the 1st European Multigrid Conference, Cologne, Germany*, Vol. 960 of: Lecture Notes in Mathematics. Springer, 1982. ISBN 0-387-11955-8.

[ISO84] International Standards Organization: *International Standard for Programming Language APL*. ISO N8485, ISO, 1984.

[ISO93] International Standards Organization: *Programming Language APL, Extended*. ISO N93.03, ISO, 1993.

[Iver62] K. E. Iverson: *A Programming Language*. John Wiley & Sons, 1st edition, 1962. ISBN 0-471-43014-5.

[Jay98] C. B. Jay: *The FISh Language Definition*. `http://www-staff.socs.uts.edu.au/~cbj/Publications/fishdef.ps.gz`, 1998.

[Jay99] C. B. Jay: *Programming in FISh*. International Journal on Software Tools for Technology Transfer, 2(3), pp. 307–315, 1999.

[JJ93] M. A. Jenkins, W. H. Jenkins: *The Q'Nial Language and Reference Manuals*. Nial Systems Ltd., Ottawa, Canada, 1993.

[Klug92] W. E. Kluge: *The Organization of Reduction, Data Flow and Control Flow Systems*. MIT Press, 1st edition, 1992. ISBN 0-262-61081-7.

[Knut97a] D. E. Knuth: *The Art of Computer Programming (Vol. 1): Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997. ISBN 0-201-89683-4.

[Knut97b] D. E. Knuth: *The Art of Computer Programming (Vol. 2): Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997. ISBN 0-201-89684-2.

[Knut98] D. E. Knuth: *The Art of Computer Programming (Vol. 3): Sorting and Searching*. Addison-Wesley, 2nd edition, 1998. ISBN 0-201-89685-0.

[KR88] B. W. Kernighan, D. M. Ritchie: *The C Programming Language*. Prentice Hall, 2nd edition, 1988. ISBN 0-13-110362-8.

[Krey98] D. Kreye: *Zur Generierung von effizient ausführbarem Code aus SAC-spezifischen Schleifenkonstrukten*. Diploma Thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1998.

[Krey02] D. Kreye: *A Compilation Scheme for a Hierarchy of Array Types*. In T. Arts, M. Mohnen (Eds.): Implementation of Functional Languages, 13th International Workshop (IFL 2001), Stockholm, Sweden, Selected Papers. Vol. 2312 of: Lecture Notes in Computer Science. Springer, 2002, pp. 18–35. ISBN 3-540-43537-9.

[Kx98] Kx Systems: *K Reference Manual Version 2.0*. Kx Systems, Miami, Florida, 1998.

[Lero02] X. Leroy: *The Objective Caml System Release 3.06*. INRIA, Rocquencourt, France, 2002.

[LRW91] M. S. Lam, E. E. Rothberg, M. E. Wolf: *The Cache Performance and Optimizations of Blocked Algorithms*. In: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '91), Santa Clara, California, USA. ACM Press, 1991, pp. 63–74.

[MSA+85] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, et al.: *Sisal: Streams and Iteration in a Single Assignment Language (Reference Manual Version 1.2)*. M 146, Lawrence Livermore National Laboratory, Livermore, California, USA, 1985.

[MTH90] R. Milner, M. Tofte, R. Harper: *The Definition of Standard ML*. MIT Press, 1st edition, 1990. ISBN 0-262-63132-6.

[OCA86] R. R. Oldehoeft, D. C. Cann, S. J. Allan: *Sisal: Initial MIMD Performance Results*. In W. Händler, et al. (Eds.): Conference on Algorithms and Hardware for Parallel Processing (CONPAR '86), Aachen, Germany. Vol. 237 of: Lecture Notes in Computer Science. Springer, 1986, pp. 120–127.

[Olde92] R. R. Oldehoeft: *Implementing Arrays in Sisal 2.0*. In: Proceedings of the 2nd Sisal Users' Conference, San Diego, California, USA. Lawrence Livermore National Laboratory, 1992, pp. 209–222.

[PAM93] S. S. Pande, D. P. Agrawal, J. Mauney: *Automatic Compiler for a Parallel Functional Language on a Distributed Memory Machine*. Technical Report, North Carolina State University, Raleigh, North Carolina, USA, 1993.

[PE01a] R. Plasmeijer, M. van Eekelen: *Concurrent Clean 1.3.1 Language Report*. University of Nijmegen, The Netherlands, 2001.

[PE01b] R. Plasmeijer, M. van Eekelen: *Concurrent Clean 2.0 Language Report (Draft)*. University of Nijmegen, The Netherlands, 2001.

[Peyt03] S. L. Peyton Jones (Ed.): *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 1st edition, 2003. ISBN 0-521-82614-4.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992. ISBN 0-521-43108-5.

[Read89] C. Reade: *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, 1st edition, 1989. ISBN 0-201-12915-9.

[Schi93] H. Schildt (Ed.): *The Annotated ANSI C Standard*. McGraw-Hill, 1st edition, 1993. ISBN 0-07-881952-0.

[Scho96] S.-B. Scholz: *Single Assignment C — Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen*. PhD Thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996. ISBN 3-8265-3138-8. Published by Shaker.

[Scho98a] S.-B. Scholz: *A Case Study: Effects of WITH-Loop-Folding on the NAS Benchmark MG in SAC*. In C. Clack, T. Davie, K. Hammond (Eds.): Implementation of Functional Languages, 10th International Workshop (IFL '98), London, England, UK, Selected Papers. Vol. 1595 of: Lecture Notes in Computer Science. Springer, 1998, pp. 216–228. ISBN 3-540-66229-4.

[Scho98b] S.-B. Scholz: *WITH-Loop-Folding in SAC — Condensing Consecutive Array Operations*. In C. Clack, T. Davie, K. Hammond (Eds.): Implementation of Functional Languages, 9th International Workshop (IFL '97), St. Andrews, Scotland, UK, Selected Papers. Vol. 1467 of: Lecture Notes in Computer Science. Springer, 1998, pp. 72–91. ISBN 3-540-64849-6.

[Scho01] S.-B. Scholz: *A Type System for Inferring Array Shapes*. In T. Arts, M. Mohnen (Eds.): Proceedings of the 13th International Workshop on the Implementation of Functional Languages (IFL 2001). Ericsson, Stockholm, Sweden, 2001, pp. 53–63.

[Scho03] S.-B. Scholz: *Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting*. Journal of Functional Programming, 13(6), pp. 1005–1059, 2003.

[SS88] S. Skedzielewski, R. J. Simpson: *A Simple Method to Remove Reference Counting in Applicative Languages*. Technical Report UCRL-100156, Lawrence Livermore National Laboratory, Livermore, California, USA, 1988.

[SSM88] V. Sarkar, S. Skedzielewski, P. Miller: *An Automatically Partitioning Compiler for Sisal*. Technical Report UCRL-98289, Lawrence Livermore National Laboratory, Livermore, California, USA, 1988.

[Stoe99] J. Stoer: *Numerische Mathematik 1*. Springer, 8th edition, 1999. ISBN 3-540-66154-9.

[Wolf89] M. J. Wolfe: *Iteration Space Tiling for Memory Hierarchies*. In G. H. Rodrigue (Ed.): Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing (PPSC '87), Los Angeles, California, USA. SIAM, 1989, pp. 357–361.