JFP **13** (6): 1005–1059, November 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796802004458 Printed in the United Kingdom

Single Assignment C: efficient support for high-level array operations in a functional setting

SVEN-BODO SCHOLZ

*Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel,
Herman-Rodewald-Strasse 3, 24118 Kiel, Germany*
(*e-mail:* sbs@informatik.uni-kiel.de)

Abstract

This paper presents a novel approach for integrating arrays with access time $\mathcal{O}(1)$ into functional languages. It introduces n-dimensional arrays combined with a type system that supports hierarchies of array types with varying shape information as well as a shape-invariant form of array comprehension called WITH-loop. Together, these constructs allow for a programming style similar to that of array programming languages such as APL. We use Single Assignment C (SAC), a functional C-variant aimed at numerical applications that is based on the proposed design, to demonstrate that programs written in that style can be compiled to code whose runtime performance is competitive with that of hand-optimized Fortran programs. However, essential prerequisites for such performance figures are a shape inference system integrated in the type system as well as several high-level optimizations. Most notably of these is *With Loop Folding*, an optimization technique for eliminating intermediate arrays.

Capsule Review

To me as a compiler writer, this work is unusually interesting because it transforms scientific code in ways that have not been considered in high-performance compilers. For example, because of the single-assignment nature of the language, loop fusion and array contraction can be done without loop alignment or data liveness analysis. The combination of novel program transformation and unique language support represents a fundamentally different alternative to traditional approaches and significantly broadens the opportunities for loop and array optimization. I believe that this work will find enthusiastic audience in the compiler community.

1 Introduction

Functional programming languages have several conceptual advantages over imperative programming languages, such as referential transparency, Church–Rosser Property, functions and complex data structures as first class objects. Nevertheless, they did not yet find a broad acceptance by application programmers outside the functional community. The reasons for this situation are manifold and differ depending on the field of application. Of primary interest in this paper are numerical applications involving complex operations on multi-dimensional arrays. In this area dominating aspects for the choice of a programming language are execution speed, support for concurrent program execution and the potential for code reuse as well as code maintenance of existing programs. On first sight, functional programming languages seem to be particularly apt for these demands. The absence of side effects allows high-level code optimizations to be applied more easily since all data dependencies are explicit. Concurrently executable subexpressions can be easily identified and the Church–Rosser Property guarantees determinate results irrespective of execution orders. Last but not least, the functional paradigm allows for a higher level of abstraction which usually improves the generality of programs and thus simplifies reuse and maintenance of existing programs.

However, in contrast to the elaborate support for lists and list operations, support for arrays with access time $\mathcal{O}(1)$ in most functional languages suffers from two shortcomings: the means for specifying general abstractions over array operations are very limited, and the apparatus that compiles/executes these specifications usually lacks suitable optimization techniques for achieving competitive runtime behavior (Hammes *et al.*, 1997).

In the context of programming languages dedicated to array processing, powerful means for specifying array operations in an abstract manner have been developed. Starting out from a mathematical notation for arrays (Iverson, 1962), several so-called *array programming languages* such as Apl (International Standards Organization, 1984), Nial (Jenkins & Jenkins, 1993), J (Burke, 1996), or K (Kx Systems, 1998) have evolved, each providing support for so-called *shape-invariant* programming, i.e. they allow arrays to be uniformly treated irrespective of their dimensionalities and their extent within the individual dimensions. Although these ideas have made their way into imperative languages such as Fortran90 (Adams *et al.*, 1992), Hpf (High Performance Fortran Forum, 1994), or Zpl (Lin, 1996), functional languages usually lack such features. Instead, most functional languages support arrays of fixed dimensionality combined with a so-called *array comprehension construct* for operations on these data structures. Although these operations can be explicitly overloaded by using type classes (Wadler & Blott, 1989), the dimensionalities of the arrays to which such operations are applicable remain restricted.

In addition to these specificational shortcomings, the integration of arrays with access time $\mathcal{O}(1)$ into functional programming languages introduces several problems concerning runtime efficiency.

In fully-fledged functional languages some slowdown is introduced by supporting functional frills such as partial application, lazy evaluation or dictionary-based realizations of overloading (Field & Harrison, 1988; Peyton Jones, 1987; Bird & Wadler, 1988; Reade, 1989; Plasmeijer & van Eekelen, 1993). Besides this loss of performance (when compared against imperative languages that usually lack such features), the particular problem of supporting arrays is to avoid superfluous creation and copying of arrays, often referred to as the 'aggregate update problem' (Hudak & Bloss, 1985; Gopinath & Hennessy, 1989; Baker, 1991).

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

Some languages (e.g. ML (Milner *et al.*, 1990) and its derivatives Caml (Leroy, 1997) and OCaml (Leroy *et al.*, 2001)) try to circumvent this problem by introducing arrays as state-full data structures and array modifications as destructive updates irrespective of the number of references that exist to them. Although this measure yields reasonable runtime efficiency, it may introduce side-effects and thus sacrifices almost all of the benefits of the functional paradigm whenever arrays are involved. Arrays are no longer first class objects and referential transparency is lost. As a consequence, highly optimizing compilers for such languages have to deal with exactly the same well-known difficulties concerning the inference of data dependencies as compilers in the imperative world (Maydan, 1992; Wolfe, 1995; Roth & Kennedy, 1996; Appel, 1998; Allen & Kennedy, 2001).

Languages that support lazy evaluation (e.g. Haskell or Clean) are facing other intricacies. Whenever strictness can not be inferred statically, it may happen that several slightly modified versions of an array have to be kept in different environments, thus leading to (temporary) space leaks. In the context of cons-lists or quad-trees (Wise, 1985; Wise, 2000) this may not be considered harmful, since parts of slightly varying data structures often can be shared within the heap. For arrays with access time $\mathcal{O}(1)$ such a sharing is impossible, which in turn renders avoiding space leaks a performance-critical issue. Even if strictness annotations or high-level optimizations such as loop fusion (Chakravarty & Keller, 2001) are utilized for reducing the number of arrays to be kept at runtime, the use of garbage collectors demands that all operations that modify arrays be implemented non-destructively. The only way to avoid such copy overhead – when sticking to garbage collection – is the use of states and state modifications in a functionally clean manner either via uniqueness types (Achten & Plasmeijer, 1993) or via state monads (Launchbury & Peyton Jones, 1994). The drawback of this solution is that the statically enforced single threading re-introduces an imperative programming style through the back door: array definitions directly correspond to array allocations in the imperative world, and whenever arrays are used as arguments for more than one modifying operation they have to be copied explicitly.

A completely different approach is taken with the functional language Sisal (McGraw *et al.*, 1985). Sisal realizes a call-by-value semantics, and the memory management of the current compiler implementation is based on reference counting. At the expense of keeping information about the number of active references at runtime, this allows to implement array operations destructively whenever possible. The performance gains easily outweight the administrative overhead for such reference counters, particularly since elaborate compiler optimizations allow to reduce this overhead to a minimum (Cann, 1989). To achieve utmost runtime efficiency, Sisal also foregoes most of the functional frills. Neither higher order functions, nor partial applications, polymorphism, or dictionary-based overloading are included in Sisal.[1] As a consequence, the Sisal compiler osc 1.3 generates code, which for some numerical benchmarks (at least in the early 1990s) outperformed

[1] This remark relates to Sisal 1.2. Although Sisal2.0 and Sisal90 both support higher order functions, to our knowledge, compilers for these Sisal dialects were never completed.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

equivalent Fortran programs in a multiprocessor environment (Oldehoeft *et al*., 1986; Cann, 1992). This demonstrates impressively that numerical algorithms can indeed benefit from the functional paradigm in terms of execution speed and suitability for concurrent execution.

However, the expressive power of Sisal does not stand out very much against that of imperative languages, such as Fortran, C, and their variants, or, more recently, Java. In Sisal, arrays are one-dimensional entities only; besides element selection only a few primitive operations such as boundary inquiries and element modifications are supported. Arrays of higher dimensionality have to be represented by nestings of such arrays, which implicitly renders the access times for individual elements dependent on the array's dimensionality. The most powerful operation on arrays is the array comprehension construct of Sisal, the so-called for-loop. It provides several compound operations such as reductions and scans over array elements which can be extracted from arrays or nestings of arrays. Despite this flexibility the expressive power of for-loops is limited by the fact that the level of nesting, i.e. the dimensionality of the argument arrays is fixed. Array operations that are applicable to arrays of any dimensionality (as they can be found in array languages such as Apl or so-called *high performance languages* such as Fortran90/Hpf) cannot be defined in Sisal.

This paper proposes a new approach towards integrating *n*-dimensional arrays with access times $\mathcal{O}(1)$ into functional languages. It is designed to allow for the specification of high-level array operations comparable to those available in state-of-the-art array programming languages on the one hand, and to facilitate a compilation to efficiently executable code with runtimes similar to those of high-performance imperative languages such as Fortran 90/Hpf, on the other hand.

The programming language of choice is called SaC (for Single Assignment C). It picks up on the design principles of Sisal but extends it by support for *n*-dimensional arrays as well as shape-invariant programming. The major design principles of SaC are

- a call-by-value semantics,
- direct support for *n*-dimensional arrays,
- support for shape-invariant array comprehensions,
- a memory management based on reference counting, and
- aggressive high-level optimizations to achieve competitive runtimes.

The kernel of SaC in fact constitutes a no-frills functional variant of C. The central idea is to stick as close as possible to the syntax of C proper, but to restrict the set of legal programs in a way that allows to define a simple mapping into the λ-calculus. The choice of a C-like syntax is motivated by two considerations. First, it allows programmers with an imperative background to have a smooth transition into the functional paradigm. Second, it facilitates the compilation process, since it allows some program parts to be mapped directly to C.

On top of this language kernel, SaC supports *n*-dimensional arrays as the major data structure. To achieve a level of abstraction similar to that of array languages

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

such as APL, J or K, all array operations can be specified shape- and thus dimension-invariantly. As shown in (Scholz, 1998b) it is this particular property which allows for an array oriented, less error-prone programming style which significantly improves program readability and code re-use. Although most other languages that lack this feature could achieve the same level of abstraction by introducing a user-defined data type, only genuine support renders effective optimizations possible (Scholz, 1998a).

Besides extended support for arrays, SAC also incorporates a state of the art module system with support for data hiding, separated name spaces, separate compilation as well as interfacing to non-SAC libraries. Based on this module system, SAC provides facilities for handling states and state modifications in a functionally sound manner based on uniqueness types. A discussion of these aspects of SAC can be found in Grelck & Scholz (1995).

The aim of this paper is not only to describe the design of SAC along with the major compilation techniques that are required for achieving runtime efficiency, but to motivate the language design, to demonstrate the programming style made possible by the design, and to show how the design is interwoven with the program optimizations actually applied.

Section 2 gives an introduction to the core language of SAC. Particular emphasis is put on the support for arrays and array operations whose introduction is accompanied by a series of small examples which demonstrate the programming flavors made possible by these language constructs.

The next two sections are of a more technical nature. They describe the type system of SAC and a scheme for compiling SAC programs into efficient C code. In particular, the type system exhibits the strong connection between language design and optimization techniques in SAC. Although the type system as it is described in section 3 primarily constitutes a part of the language description, its design is guided by the intent to use type inference for statically inferring array shapes which turns out to be a prerequisite for many advanced optimization techniques. The most challenging design problem in this context is to let shape inference accept all shape correct programs without making the type system undecidable.

A solution to these conflicting aims is given in section 3. It is based on the idea to make shapes (and thus shape inference) an optional part of the type system rather than requiring exact shapes for deciding typeability. This is achieved by the introduction of a hierarchy of array types accompanied by dynamic type checks which may be inserted during type inference. As a consequence, the type system turns out to be ambiguous wrt. the shape information, i.e. the extent to which shapes are statically inferred does not depend on the type system but on the implementation of the type inference algorithm actually applied.

Section 4 describes the actual compiler implementation. In the first part of it an overview of the compilation process is given, including a description of the type inference algorithm actually applied. The remaining parts of section 4 focus on the optimization and compilation techniques that relate directly to the design choices concerning arrays and array operations as introduced in section 2.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

The interplay between language design, programming style, and compilation techniques is summarized by means of a case study in section 5. It discusses several different SaC implementations of a numerical benchmark and contrasts them with an Hpf implementation wrt. programming style and runtime efficiency. Readers who are primarily interested in the language design and a comparison with languages such as Hpf may want to first have a look at this section prior to reading the technical details of sections 3 and 4, as only parts of these sections are required to follow the expositions made.

Section 6 concludes the paper and points out future research directions.

2 The basics of SaC

This section gives an introduction to the basic design of SaC. It includes just the bare essentials that are necessary to write useful programs, and then focuses in some detail on the array concept supported by SaC.

2.1 A functional subset of C

Identifying a functional subset of a classical imperative language such as C immediately raises the question of what exactly are the essential differences between the functional and the imperative programming paradigm.

As the semantics of functional languages are based on the λ-calculus (or a combinatory calculus) (Plotkin, 1974; Barendregt, 1981; Hindley & Seldin, 1986), program execution may conceptually be considered a process of meaning-preserving program transformations governed by a set of non-overlapping and context-free rewrite rules which systematically substitute equals by equals until no more rules are applicable (Turner, 1979; Kluge, 1992; Plasmeijer & van Eekelen, 1993). Such context-free substitutions are responsible for two closely interrelated properties. One is referential transparency which ensures that the value of an expression is solely determined by the expression itself, irrespective of the context in which it is being evaluated, the other is the Church–Rosser property which ensures that this value, apart from termination properties, is invariant against execution orders. Put another way: functional languages are free of side effecting operations which would violate these properties.

In contrast, side effects are the very essence of the imperative model, which is based on the concept of states and step-wise state transformations, i.e. programs are designed to perform (side) effects on states.

The problem of defining a functional subset of C then obviously boils down to systematically eliminating side-effects.

The primary source of (intended) side effects in C programs are functions (many of which should more properly be called procedures). They may not only be used to compute values but also to effect changes in their calling environments through assignments made in the function bodies either to global variables or to reference parameters. Thus, a major step towards turning C into a functional language simply consists in outlawing global variables and reference parameters, and hence

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

```
Program      ⇒  [ FunDef ]* Type main ( ) ExprBlock
FunDef       ⇒  Type FunId ( [ ArgDef ] ) ExprBlock
ArgDef       ⇒  Type Id [ , Type Id ]*
ExprBlock    ⇒  { [ Vardec ]* [ Assign ]* RetAssign }
             |  { [ Vardec ]* SelAssign RetAssign }
Vardec       ⇒  Type Id ;
Assign       ⇒  Id = Expr ;
RetAssign    ⇒  return ( Expr ) ;
SelAssign    ⇒  if ( Expr ) AssignBlock else AssignBlock
AssignBlock  ⇒  { [ Assign ]* }
Expr         ⇒  ( Expr )
             |  Id ( [ Expr [ , Expr ]* ] )
             |  Expr Prf Expr
             |  Const
             |  Id
Prf          ⇒  + | - | * | / | == | !=
             |  < | <= | > | >=
Type         ⇒  int | float | double | char
```

Fig. 1. The kernel of SaC.

pointers. As a nice aside, the functions that abide by these restrictions are in fact (super)combinators since C also outlaws nested function declarations.

Another problem appear to be multiple assignments to local variables declared inside the bodies (statement blocks) of C functions. However, considering assignments as the equivalent of introducing let-bound variables, multiple assignments to the same variable may simply be viewed (and treated) as nestings of let constructs of which each defines a new binding which shadows the preceding one, thus giving the entire statement block a perfectly functional interpretation.

IF-THEN-ELSE clauses, contrary to often heard arguments, do not pose a problem either. Functions in which they are top level, making up the entire body, may return values assigned to any of the local variables defined in both the consequent and the alternative block. If they are not top level, they may simply be considered (and actually transformed into) functions (abstractions) of the variables defined in the surrounding block(s) which are applied to instantiations of these variables.

Likewise, loop constructs may be considered (and transformed into) tail-recursive functions and, in their syntactical positions, replaced by the respective function calls. Thus, ruling out global variables and pointers seems to be doing the trick of extracting from C a computationally complete functional subset which may be taken as the kernel language of SaC.

Figure 1 shows that part of the kernel language that is relevant for the remainder of the text.

As in C, SaC programs consist of a sequence of function definitions, the last of which is a designated function `main`. The syntax of function headers is also adapted from C. Function bodies are merely restricted to sequences of assignments terminated by a return expression and optionally preceded by type declarations for variables. The only alternative to such sequences of assignments is a single top-level IF-THEN-ELSE clause with assignments in the two alternative branches, and a return expression directly following the IF-THEN-ELSE clause. In contrast to C where the right hand side of an assignment may contain further assignments, in SAC right-hand sides of assignments are restricted to expressions, i.e. to arbitrary nestings of function applications, variables and constants.

It should be noted here that, as a consequence of dropping pointers, this kernel does not include any non-scalar data structures as they are known from C. Instead, SAC comes with a high level array concept (to be introduced in the next section) which is completely compatible with the functional paradigm, i.e. functions conceptually consume entire argument arrays and produce new result arrays.

2.2 The array concept of SAC

SAC supports the notion of n-dimensional arrays and of high-level array operations as they are known from array languages such as APL (Iverson, 1962; International Standards Organization, 1984), NIAL (Jenkins & Jenkins, 1993) or J (Burke, 1996). As of now, arrays are the sole data structures in SAC; for reasons of uniformity even scalar values are considered arrays. An array is represented by a data vector $[d_0, \ldots, d_{q-1}]$ which contains its elements, and by a shape vector $[s_0, \ldots, s_{n-1}]$ which defines its structure, with n and s_i respectively specifying the number of axes (or the dimensionality) and the number of elements (or the index range) along the i-th axis of the array. Data and shape vectors cannot be entirely freely chosen, but must satisfy the equation $q = \prod_{i=0}^{n-1} s_i$.

Given these two entities, an n-dimensional array a may in SAC be specified as an expression of the form

$$\texttt{reshape(}\ [s_0, ..., s_{n-1}]\texttt{,}\ \ [d_0, ..., d_{q-1}]\texttt{)}$$

where reshape is a built-in primitive, n, $q \in \mathbb{N}_0$, all s_i are non-negative integer values,[2] and all d_j are atomic (scalar) values of the same type. The special cases of scalars and vectors may be denoted as

$$\begin{array}{rcll} s & \equiv & \texttt{reshape([], [}s\texttt{])} & \text{, and} \\ [v_0, ..., v_{n-1}] & \equiv & \texttt{reshape([}n\texttt{], [}v_0, ..., v_{n-1}\texttt{])} & \end{array}$$

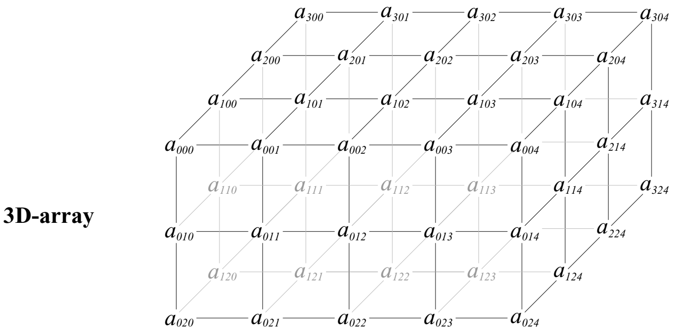
Subarrays or elements of an n-dimensional array of shape $[s_0, \ldots, s_{n-1}]$ may be accessed (or addressed) by index vectors from the set

$$\begin{array}{rl} \mathscr{L}iv(\,[s_0, ..., s_{n-1}]) := \{\,[iv_0,\ ..., iv_{m-1}] \mid & 0 \leqslant m \leqslant n, \\ & 0 \leqslant iv_0 < s_0,\ ..., 0 \leqslant iv_{m-1} < s_{m-1}\} \end{array}$$

which will be referred to as the set of legitimate index vectors. It should be noted

[2] The alert reader may note here that this introduces infinitely many distinct empty arrays that vary in their non-zero shape components only. For a discussion of the implications of this design decision, see Jenkins & Glasgow (1989) and Jenkins (1999).

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press


3D-array
a_{300} a_{301} a_{302} a_{303} a_{304}
a_{200} a_{201} a_{202} a_{203} a_{204}
a_{100} a_{101} a_{102} a_{103} a_{104} a_{314}
a_{000} a_{001} a_{002} a_{003} a_{004} a_{214}
a_{110} a_{111} a_{112} a_{113} a_{114} a_{324}
a_{010} a_{011} a_{012} a_{013} a_{014} a_{224}
a_{120} a_{121} a_{122} a_{123} a_{124}
a_{020} a_{021} a_{022} a_{023} a_{024}

shape vector: *[4, 3, 5]*
data vector: *[a_{000}, a_{001}, ... , a_{004}, a_{010}, ... , a_{014}, ... , a_{024},*
a_{100}, a_{101}, ... , a_{124},
a_{200}, a_{201}, ... , a_{224},
a_{300}, a_{301}, ... , a_{324}]

Fig. 2. Representing arrays by shape and data vectors.

here that this definition does not only comprise index vectors with less components than the dimensionality of the array to be accessed, but it also always includes the empty index vector.

Let a be an array as defined above and let $\texttt{iv}=[iv_0, ..., iv_{m-1}] \in \mathscr{L}iv([s_0, ..., s_{n-1}])$ be a legitimate index vector in a, then an expression `a[ iv ]` refers to the subarray

$$\texttt{reshape}([s_m, ..., s_{n-1}], [d_p, ..., d_{p+l-1}])$$

where $p = \sum_{j=0}^{m-1} \left(iv_j * \prod_{k=j+1}^{n-1} s_k \right)$ and $l = \prod_{i=m}^{n-1} s_i$.

The special case $m = n$ specifies selection of individual array elements, as the index vector $[iv_0, \ldots, iv_{m-1}]$ refers to the subarray $\texttt{reshape}([], [d_p])$, i.e. to the scalar value d_p.

As an example, figure 2 shows a three-dimensional array with shape and data vectors as given underneath. An index vector of, say `[0]`, refers to the two-dimensional array shown as the front-most plane of the cube, i.e. the array `reshape([3,5],` $[a_{000}, \ldots, a_{024}]$`)`. An index vector of, say `[1,0]`, refers to the second row vector of the topmost horizontal plane, i.e. the vector `reshape([5],`$[a_{100}, \ldots, a_{104}]$`)`, and an index vector of, say `[3,0,4]`, picks the rightmost element of the last row vector in the topmost horizontal plane, i.e. the scalar `reshape([],`$[a_{304}]$`)`.

2.3 Using compound array operations

Similar to other array languages such as APL or FORTRAN90, SAC suggests using so-called *compound array operations* which apply uniformly to all array elements or to the elements of coherent subarrays. All these operations are defined shape-invariantly, i.e. they can be applied to arrays of arbitrary shape and thus to arrays of arbitrary dimensionality.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

The following introduces the most important compound array operations available in the standard library of the actual SaC compiler.

Let *a* and *b* be SaC expressions that evaluate to arrays, let *v* be a SaC expression that evaluates to a vector of non-negative integers, and let *iv* denote an expression that evaluates to a legitimate index vector of array *a*, then the expression

`dim(` *a*`)` returns the dimensionality, i.e. the number of axes, of *a*;

`shape(` *a*`)` returns the shape vector of *a*;

`sel(` *a*, *iv*`)` $\equiv$ ***a*[*iv*]** returns the sub-array of *a* selected by *iv*;

`genarray(` *v*, *val*`)` returns an array whose shape vector is identical to the concatenation of *v* and `shape(` *val*`)`; it's data vector is composed of repeated copies of the data vector of *val*;

`modarray(` *a*, *iv*, *val*`)` returns a new array which differs from *a* in that the subarray *a*`[`*iv*`]` is being replaced by the array *val*, provided that `shape(` *val*`)` $=$ `shape(` *a*`[`*iv*`])`;

`take(` *v*, *a*`)` returns an array *r* with `shape(`*r*`)` $= v$ and *r*`[`*i*`]` $=$ *a*`[`*i*`]` for all index vectors $i \in \{$ `[0,...,0]`, ..., $[v_0 - 1, ..., v_{n-1} - 1]$ $\}$, provided that $[0, ..., 0] \leqslant v = [v_0, ..., v_{n-1}] \leqslant$ `shape(`*a*`)` component-wise, otherwise it is undefined;

`drop(` *v*, *a*`)` returns an array *r* with `shape(`*r*`)` $= [shp_0, ..., shp_{n-1}]$ where $shp_i =$ `(shape(`*a*`)[`*i*`] -` *v*`[`*i*`])` for all $i \in \{$`[0],...,[n-1]`$\}$, and *r*`[`*j*`]` $=$ *a*`[`$v \vec{+} j$`]` for all index vectors $j \in \{$`[0,...,0]`, ..., $[shp_0, ..., shp_{n-1}]\}$ (where $\vec{+}$ denotes element-wise addition of vectors), provided that $[0, ..., 0] \leqslant v = [v_0, ..., v_{n-1}] \leqslant$ `shape(` *a*`)` component-wise, otherwise it is undefined;

`cat(` *d*, *a*, *b*`)` catenates the arrays *a* and *b* along their d^{th} axis if the shapes along the other axes are identical, otherwise it is undefined.

In addition to these structuring operations, the standard library of SaC also includes all binary arithmetic, logic and relational (or value-transforming) operations of C. They are not only applicable to scalar values but also element-wise to argument arrays, provided both are either of the same non-scalar shape or one of them is a scalar.

As an example that demonstrates the expressive power of such compound array operations over explicit nestings of loops that traverse individual array elements in some suitable order, consider the special case of matrix vector multiplication, where the matrix contains non-zero elements along three diagonals only, as shown schematically in figure 3. Here, the three diagonals are located symmetrically to the main diagonal with a distance of `offset` elements along the matrix rows. They may be represented by three vectors `d0, d1` and `d2`. Sparing the multiplications with the 0 elements outside of the three diagonals, the matrix vector product for each element of the result vector requires only two or three product terms to be summed up (cf. right-hand side of figure 3).

A conventional implementation consists of three consecutive FOR-loops, each of which modifies a pre-allocated section of the result vector whose elements can in turn be computed uniformly by the same parameterized arithmetic expression:

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

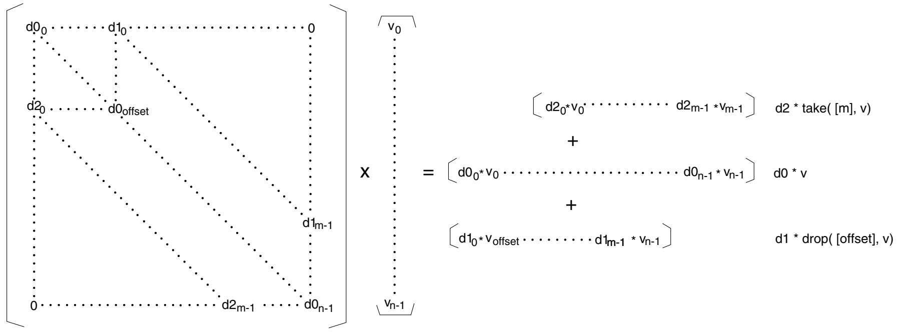


Fig. 3. Multiplying a sparse matrix with a vector.

```
{ ..
  u = genarray( [n], 0.0);

  for( i=0; i < offset; i++)
    u = modarray( u, [i], d0[[i]] * v[[i]]
                          + d1[[i]] * v[[i+offset]]);
  for(; i < n-offset; i++)
    u = modarray( u, [i], d0[[i]] * v[[i]]
                          + d1[[i]] * v[[i+offset]]
                          + d2[[i-offset]] * v[[i-offset]]);
  for(; i < n; i++)
    u = modarray( u, [i], d0[[i]] * v[[i]]
                          + d2[[i-offset]] * v[[i-offset]]);
 .. }
```

The explicit indexing of vector elements in this rather elaborate piece of code requires the loop boundaries and index offsets to be chosen very carefully, which is known to be a very error-prone undertaking. Moreover, it is not at all obvious that the indexing of elements reflects the intended functionality.

This contrasts with a much more concise specification which consequently uses compound array operations:

```
{ ..
  zeros = genarray( [offset], 0.0);

  u = d0 * v;
  u += cat( 0, zeros, d2 * take( [m], v));
  u += cat( 0, d1 * drop( [offset], v), zeros);
 .. }
```

This code does completely without explicit indexing through vector elements. Instead, the indexing is hidden inside the overloaded versions of `*` and `+`. It also exposes more clearly the contribution of the three diagonals to the result.

Since `*` and `+` both require their arguments to be of the same size, the smaller vectors `d2 * take( [m], v)` and `d1 * drop( [offset], v)` have to be prepanded or appended by vectors of zeros, which introduces some small amount of redundancy in the specification.

2.4 The WITH-loop construct

As illustrated in the preceding subsection, programming array computations can be made decidedly more concise, comprehensible, and less susceptible to errors using combinations of compound array operations rather than (nestings of) tailor-made loops that traverse array elements with specific starts, stops and strides. However, choosing a set of compound operations which is universally applicable and also suits different programming styles seems to be a problem that can hardly be solved satisfactorily. Though several such comprehensive sets have evolved in the course of developing APL and other array languages, there are still ongoing discussions about potential extensions (Lucas, 2001).

Specifying within the language itself new from existing compound operations to serve the specific needs of a particular application area often turns out to be a very diffcult undertaking. It usually involves decomposing arrays into sub-arrays, possibly even down to the level of elements, and re-assembling them in a different form. The resulting program terms tend to be rather complex, difficult to understand, and to some extent are often burdened with redundant operations.

An example in kind is the addition of two vectors of different lengths in the SAC code fragment of the preceding subsection. Since the operator + is only defined on vectors of the same lengths, vectors of zeros have to be catenated to the smaller vectors in order to adjust the vector sizes. Without falling back on an element-wise specification this redundancy can be avoided by a new compound operation

`EmbeddedAdd(` *`small`*`,` *`offset`*`,` *`large`*`)` which adds the elements of an array *`small`* to the elements of an array *large* that are within the index range { *`offset`*`, ...,` *`offset`* `+ shape(` *`small`*`) - 1` }, provided that *`offset`* `+ shape(` *`small`*`)` ⩽ `shape(` *`large`*`)` holds componentwise.

Defining such an operation in a shape-invariant style becomes quite an artful piece of programming if only the compound operations of the standard SAC library as defined in the preceding subsection are available. The problem is that for each axis (dimension) an array of zeros has to be created and subsequently catenated to the smaller array, which results in a fairly complex specification.

The approach taken in SAC as the way out of this dilemma is to provide some sufficiently versatile language construct which allows to specify in concise and efficiently executable form shape-invariant compound operations which are tailored to the specific needs of some application problem at hand. The construct introduced for this purpose is a so-called WITH-loop. It may be considered a generalization of array comprehensions which, in contrast to those known from other functional languages such as Haskell, CLEAN, or SISAL, can be specified in shape-invariant form. Using WITH-loops suitable application-specific compound operations (such as `EmbeddedAdd`) may be freely defined with little effort and added to the standard library. Application programs may thus be composed almost entirely of compound operations, liberating program design from tedious, hard to understand and error-prone specifications of starts, stops and strides of traversals over index ranges.

The most important benefits come with the implementation of WITH-loops. Since they are general enough to define almost all high-level array operations

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

```
Expr            ⇒  ...
                |  with ( Generator ) Operation

Generator       ⇒  Bound Rel Id Rel Bound [ Filter ]
Bound           ⇒  Expr | .
Rel             ⇒  <= | <
Filter          ⇒  step Expr [ width Expr ]

Operation       ⇒  [ { LocalDeclarations } ] ConExpr
ConExpr         ⇒  genarray ( Expr , Expr )
                |  modarray ( Expr , Expr , Expr )
                |  fold ( FoldFun , Expr , Expr )
FoldFun         ⇒  + | * | Id
```

Fig. 4. WITH-loops in SAC.

either directly as WITH-loop based primitives or as compositions of several such primitives, all compiler optimizations concerning efficient code generation for array operations can be concentrated on this single construct. Particularly rewarding in terms of performance gains is an optimization technique called *With Loop Folding*. It completely eliminates in almost all cases the generation of intermediate arrays between successive value-transforming or (re-)structuring operations. This technique, which for many application programs tips the performance scale in favor of SAC, takes full advantage of the referential transparency that comes with the underlying functional model; it is not generally applicable in the context of imperative array processing languages such as FORTRAN90 or APL.

WITH-loops consist of a generator part which specifies an index vector set and an operator part which specifies an operation to be applied to the elements of the index vector set. They come in three forms which differ from each other with respect to the operator part, these being

- *genarray* WITH-loops which create arrays from scratch;
- *modarray* WITH-loops which conceptually produce new arrays from existing ones and, in doing so, modify their entries or the entries of some subarrays;
- *foldarray* WITH-loops which fold by means of some binary operation into single values expressions computed over the range of indices specified by the generator part, which in many cases are (but need not be) just the elements of an array.

The syntax of WITH-loops is specified in figure 4. WITH-loops are SAC expressions preceded by the keyword (or constructor) `with` followed by the generator and the operator expressions. The generator defines a set of index vectors by means of two boundary expressions. They must evaluate to two vectors of identical length which componentwise specify the maximum and the minimum indices of the index vector set defined. Depending on its syntactical position, the symbol '.' is used as short hand for the lowermost or highermost legal index vector of the array to be

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

generated/modified by the entire WITH-loop.[3] The generator also includes a `step` vector which may be used to define strides larger than one over the index vector range, and a `width` vector which may be used to define larger recurring sections to which the operation part must be applied. Formally, a generator `(` l `<= iv <=` u `step` s `width` w`)`, where l, u, s and w are vectors of length $n \in \mathbb{N}_0$, defines the index vector set

$$\mathcal{G}en(\,l,u,s,w) := \{[iv_0, ..., iv_{n-1}] \mid \forall_{j \in \{0,...,n-1\}} : \quad l_j \leqslant iv_j \leqslant u_j \\ \wedge\ (iv_j - l_j) \text{ modulo } s_j < w_j\}$$

The operator part is composed of optional local variable definitions followed by a constructor expression which defines one of the three WITH-loop alternatives.

The `genarray`-constructor expects two arguments. Its first argument specifies the shape of the result, and the second argument specifies the values of the elements (subarrays) whose indices are defined by the generator part. It may be parameterized by the identifier specified in the generator part which stands in for the index vector in consideration. All other elements are initialized with 0. Thus, a WITH-loop

```
with( l <= iv <= u step s width w)
genarray( shp, expr(iv))
```

computes an array a with

$$\begin{array}{lll} \texttt{shape(}\ a\texttt{)} & := & \texttt{cat(0, } shp\texttt{, shape(}expr\texttt{(iv)))} \\ a\texttt{[iv]} & := & \left\{ \begin{array}{ll} expr\texttt{(iv)} & \text{for } \texttt{iv} \in \mathcal{G}en(l,u,s,w) \\ \texttt{genarray(shape(}expr\texttt{(iv)), 0)} & \text{otherwise} \end{array} \right. \end{array}$$

provided that $\mathcal{G}en(l,u,s,w) \subseteq \mathcal{L}iv(\,shp)$.

The `modarray`-constructor requires three arguments: an array to be modified, the actual index vector (usually the identifier of the generator), and the value to be inserted into this index position. Again, only the elements (subarrays) that are referred to by the index vectors of the generator are affected. More formally, a WITH-loop

```
with( l <= iv <= u step s width w)
modarray( array, iv, expr(iv))
```

computes an array a with

$$\begin{array}{lll} \texttt{shape(}\ a\texttt{)} & := & \texttt{shape(}\ array\texttt{)} \\ a\texttt{[iv]} & := & \left\{ \begin{array}{ll} expr\texttt{(iv)} & \text{for } \texttt{iv} \in \mathcal{G}en(l,u,s,w) \\ array\texttt{[iv]} & \text{otherwise} \end{array} \right. \end{array}$$

provided that $\mathcal{G}en(l,u,s,w) \subseteq \mathcal{L}iv(\,shp)$ and $\texttt{shape(}expr(iv)\texttt{)=shape(}array\texttt{[iv])}$ for all $iv \in \mathcal{G}en(l,u,s,w)$.

Finally, the `fold`-constructor requires as arguments a binary commutative and associative fold function, its neutral element, and an expression to be folded over the index range specified by the generator. Then, a WITH-loop

```
with( l <= iv <= u step s width w)
fold( fun, neutral, expr(iv))
```

[3] Note here that the symbol '.' may not be used in the context of *foldarray* WITH-loops.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

computes an expression a with

$$a := \begin{cases} neutral & \text{iff } \mathscr{G}en(l,u,s,w) = \emptyset \\ neutral\ \ fun\ \ expr(iv^0)\dots fun\ \ expr(iv^{q-1}) & \text{otherwise} \\ \qquad \text{where } \{iv^0 \dots iv^{q-1}\} = \mathscr{G}en(l,u,s,w) & \end{cases}$$

It should be noted here that fun is used in infix notation without any brackets in order to emphasize that the order in which the expressions are folded is intentionally left unspecified. This allows for an arbitrary evaluation order to be chosen by the compiler which, among other advantages, greatly facilitates code generation for non-sequential execution. However, it is the responsibility of the programmer to make sure that the fold function is commutative and associative in order to guarantee deterministic results.

2.5 *Defining compound array operations using* WITH*-loops*

Having WITH-loops available, a function `EmbeddedAdd` as introduced in the previous subsection may be rather elegantly defined as:

```
double[*] EmbeddedAdd( double[*] small, int[.] offset, double[*] large)
{
  res = with( offset <= iv < offset + shape(small))
        modarray( large, iv, large[iv] + small[iv-offset]);
  return( res);
}
```

This function merely consists of a single WITH-loop-construct. It computes an array that differs from the third argument `large` only in the index range {`offset`, ..., `offset + shape( small) -1` }. Within this index range the elements of the first argument `small` are added to those of the array `large`. The type declarations `double[*]` and `int[*]` denote arrays of unspecified shape whose elements are of type `double` and `int`, respectively, and `int[.]` declares a vector of integers.[4] The most important aspect of this specification is its shape-invariance. It is achieved by defining the result shape and the index vector range of the generator in terms of the shapes of the argument arrays.

Another example for the expressive power of WITH-loops can be found in the standard library of the actual SAC compiler release,[5] a generalized version of the `take` operation defined in section 2.3:

```
double[*] take( int[.] v, double[*] a)
{
  res = with( . <= iv <= .)
        genarray( v, a[iv]);
  return( res);
}
```

[4] The full treatment of the SAC type system can be found in the next section.
[5] see <http://www.informatik.uni-kiel.de/~sacbase/>.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

The WITH-loop used here has an interesting property which may not be obvious on first glance. Assuming that v is of maximum length, i.e. (shape(v)=[dim(a)]), the function has exactly the functionality defined in section 2.3.

However, the WITH-loop also also yields useful results if (shape(v) < [dim(a)]). Assuming that shape(a) = [$s_0 \ldots s_{n-1}$] and shape(v) = $[m] < [n]$, the index vectors from the interval (0*v <= iv <= v-1) all refer to subarrays of a with shape [$s_m \ldots s_{n-1}$]. In other words, for the missing components of v (wrt. the dimensionality of a) all elements are selected, i.e.

take(v, a) = take(cat(0, v, [$s_m \ldots s_{n-1}$]), a) .

Since all (re-)structuring and value-transforming primitives introduced in the preceding subsection (and many more) can be specified as library functions in a similar way, the only primitives left that must be implemented as built-ins are dim, shape, sel and reshape.

3 The type system of SAC

As for most statically typed languages, the type system of SAC primarily serves three purposes. It must be capable of

- statically detecting potential runtime errors that result from applications of partially defined functions to arguments that are not within their domains,
- supporting the specification of domain restrictions to improve program program readability and program extensibility,
- and, last not least, providing argument information that helps to vastly improve the code generation wrt. runtime efficiency.

The extent to which these objectives are met critically depends on the granularity of the types available. The more specific the types that are supported the more detailed analyses can be made by the type system. In the context of arrays and array operations it is important for the type system to be capable of statically detecting shape incompatibilities, to restrict function domains to particular argument shapes, and, most importantly, to generate shape specific code.

The latter is an essential prerequisite for the generation of efficiently executable code. For instance, if scalars (0-dimensional arrays) cannot be identified statically, at runtime, they have to be allocated in the heap rather than on the stack, thus, introducing considerable overhead. Likewise, code generation for WITH-loops significantly profits from static knowledge of array shapes. Only if the shapes of the index vectors of a WITH-loop are statically known, efficient code consisting of one loop per dimension can be generated. Otherwise, more generic code is required that iterates through the components of the index vectors provided by the generator part.

Unfortunately, shape-specific types conflict with the idea of shape-invariant specifications since they require array types without shape restrictions. To overcome this problem the type system of SAC introduces for each element type a hierarchy of array types containing varying levels of shape information: no shape information at all, information about the dimensionality only, or the exact shape. Based on these type hierarchies, the type system provides rules for inferring types as specific as possible.

Starting from the main function, shape information is propagated from outermost to innermost. This propagation of shapes requires the types of shape-invariant functions to be refined whenever an application to statically known shapes is met.

However, this approximation of function types has some subtle limitations. The problems involved can be demonstrated by means of three simple shape-invariant functions:

```
int[*] id( int[*] a)     int[*] dupl( int[*] a)         int[*] foo( int[*] a)
{                        {                              {
  return( a);              return( cat( 0, a, a));        if( pred( a))
}                        }                                  res = a;
                                                          else
                                                            res = [1];
                                                          return( res);
                                                        }
```

where INT[*] refers to integer arrays of arbitrary shape and pred is assumed to be of type INT[*] → BOOL.

All three functions expect an integer array a as argument and compute a new one: id simply returns a, dupl 'duplicates' a by catenating it to itself wrt. the outermost dimension, and foo either returns a or the vector [1], depending on the value of some predicate function pred applied to a.

In fact, id, dupl, and foo are representatives for three different classes of shape-invariant array operations which require increasingly elaborate extensions of the type system to infer result shapes from given argument shapes.

The simplest of these shape-invariant array operations, id, returns arrays of the same shape as one of their arguments. Hence, support for polymorphism would suffice to infer exact return shapes. The function dupl is a representative for the majority of shape-invariant array operations. The shapes of their return values depend in more complicated ways on argument shapes without being identical to one of them. Functions such as foo are even worse. Since their result shapes depend on argument values rather than shapes, it requires dependent types to figure out exact result shapes at compile time.

Supporting dependent types (Martin-Löf, 1980) as for example done in Cayenne (Augustsson, 1998) comes at the cost of undecidability on the one hand and the need for complex type specifications on the other hand. Decidability can only be regained by restricting the dependent types. Examples for this approach are the so-called *indexed types* in Zenger (1998), or the so-called *sized types* in Hughes *et al.* (1996) and Chin & Khoo (2000). The drawback of these solutions is that they restrict the legal shape dependencies to guarantee termination. Furthermore, much more complex type specifications are required to successfully type functions such as dupl or foo, causing specificational redundancy. Another related approach which faces the same limitations is the shape inference in Fish (Jay & Steckler, 1998). Though the shape specifications of Fish are not called types or type specifications, wrt. shape inference, they serve the same purpose and they use the same mechanisms as dependent types do.

$$\begin{array}{lcl} Type & \Rightarrow & ElemType\ \texttt{[}\ \big[\,ShpSpec\,\big]\ \texttt{]} \\ & | & ElemType \\ ShpSpec & \Rightarrow & Num\ \big[\ ,\ Num\ \big]^{*} \\ & | & .\ \big[\ ,\ .\ \big]^{*} \\ & | & \texttt{*} \\ & | & \texttt{+} \\ ElemType & \Rightarrow & \texttt{int}\ |\ \texttt{float}\ |\ \texttt{double}\ |\ \texttt{char}\ |\ \texttt{bool} \end{array}$$

Fig. 5. Array types in SaC.

Since neither a restriction of potential shape dependencies nor complex type annotations suit the idea of elegant Apl-like specifications well, the approach taken in SaC is based on a combination of dynamic type checks and a mechanism similar to multi-parameter type classes with default instances, which at least for functions of the first two categories (`id` and `dupl`) allows exact result shapes to be inferred.

The basic idea is to consider a shape-invariant function definition as a combination of a class definition and the definition of a default instance of it. During shape inference, these default instances can be specialized to specific shapes whenever applications to arrays of known shapes are found.

For some expressions, e.g. applications of `foo`, this approach fails to infer exact result shapes though all argument shapes are available. If such expressions serve as arguments to functions that are defined for a particular shape only, shape conformity can not be statically guaranteed. In order to prevent the type system from rejecting such programs, it must support so-called *type assertions*, which postpone conformity checks until runtime. In fact, these assertions can be inserted by the type inference system itself whenever a subtype of an actually inferred type is required to successfully type a program.

3.1 The hierarchy of array types in SaC

SaC provides for each element type an entire hierarchy of array types which contains different levels of shape information. The most general type, which primarily is needed for shape-invariant specifications, does not contain any shape information at all. More restricted types either prescribe a certain dimensionality or a specific shape. The syntax of array types in SaC is given in figure 5. An array type consists of an element type followed by a shape specification. For reasons of compatibility, not only the element types are adopted from C but the shape specifications for 0-dimensional arrays (scalars) may be omitted as well. If the shape is to be given precisely, the shape specification simply consists of a shape vector. The components of such a shape vector may be replaced by wildcards that refer to less specific types. Arrays of known dimensionality are denoted by vectors of '.' – symbols whose lengths indicate the dimensionalities. If even the number of dimensions is to be left unspecified, the dots may be replaced by either a '+' or a '*', indicating arrays of at least dimensionality one or arbitrary dimensionality, respectively.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

Using $\preceq$ as notation for subtyping the following reflexive and transitive relations for all element types τ, $n \in \mathbb{N}$, $s_1, \ldots, s_n \in \mathbb{N}_0$ hold:

$$
\begin{array}{ll}
\text{(scalars)} & \tau \equiv \tau[\,] \preceq \tau[*] \\
\text{(non-scalars)} & \tau[s_0, ..., s_{n-1}] \preceq \tau[\overbrace{\bullet, ..., \bullet}^{n}] \preceq \tau[+] \preceq \tau[*]
\end{array}
$$

3.2 *The type rules of* SaC

The basic type rules of SaC are similar to those of a standard monomorphic type system (e.g. Reade, 1989). One of the key extensions derives from the differences in treating variables in C and in SaC. Whereas a variable in C identifies a box whose value may change at runtime, in SaC, an assignment constitutes a let expression which introduces a new variable. To allow the programmer to switch between these two different concepts, the type system restricts the types of identically named variables within function bodies to have at least a common supertype.

The type environments used to define the type system of SaC consist of tuples of the form (var : $<\delta,\tau>$), where δ denotes the so-called *declaration type* of an identifier var, and τ denotes the so-called *actual type* of var. Modifications of a type environment A are denoted as

$$
A\{v : <\delta, \tau>\} := \begin{cases} A \setminus \{(v : <\delta', \sigma>)\} \cup \{(v : <\delta, \tau>)\} & \text{iff } \exists (v : <\delta', \sigma>) \in A \\ A \cup \{(v : <\delta, \tau>)\} & \text{otherwise} \end{cases}
$$

Furthermore, $A \vdash e : \tau$ denotes an assertion that in a type environment A an expression e has type τ. With these definitions at hand, the basic set of type rules for SaC can be defined as in figure 6. It includes all rules for constructs that appear typically in function bodies but not those for user-defined or primitive functions. Assuming that f_{Type} for SaC constants (i.e. homogeneous arrays) computes their (most specific) type, the basic rules Const, Var, and Return are straightforward. The VarDec rule makes sure that (i) the declaration of a variable in the beginning of a function body complies its usage in the remainder of the body, and that (ii) there is one declaration per identifier at most.

The Let rule realizes the restriction imposed on variables mentioned above. Whenever there exists a declaration for a variable v in A, i.e. $\exists$ (v : $<\delta,\sigma'>$) $\in$A, the type of the expression on the right-hand side has to be a subtype of the declaration type. Otherwise, a declaration for v is added, assuming the most general type that includes the type of the actual right-hand side. To do so, a function f_{BaseType} is used, which computes the element type component of a given SaC type.

The rule for treating conditionals also is straightforward: if typing the two different branches yields two different result types, they have to have a common supertype and their least upper bound (f_{lub}) serves as overall result type.

The last two rules concern type assertions of the form assert(e, τ). The rule assert1 ensures that type assertions are accepted if the actual type inferred for the expression is a supertype of the asserted type. If the inferred type turns out to be even more specific than the asserted type, the assertion may be eliminated and type inference may continue with the more specific type (rule assert2).

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

$$\textsc{Const} \quad : \quad \frac{}{A \vdash \mathit{Const} : f_{\mathrm{Type}}(\mathit{Const})}$$

$$\textsc{Var} \quad : \quad \frac{}{A \vdash v : \tau} \qquad \text{if} \qquad (v : <\delta, \tau>) \in A$$

$$\textsc{Return} \quad : \quad \frac{A \vdash e : \tau}{A \vdash \textsc{return}(e) : \tau}$$

$$\textsc{VarDec} \quad : \quad \frac{A\{v : <\delta, \delta>\} \vdash \mathit{Rest} : \tau \quad A \vdash \mathit{Rest} : \tau}{A \vdash \delta\ v;\ \mathit{Rest} : \tau}$$

$$\text{if } \neg\exists\ (v : <\delta', \delta'>) \in A$$

$$\textsc{Let} \quad : \quad \frac{A \vdash e : \sigma \quad A\{v : <\delta, \sigma>\} \vdash \mathit{Rest} : \tau}{A \vdash v = e;\ \mathit{Rest} : \tau}$$

$$\begin{aligned} &\text{if } (\ \exists\ (\mathrm{v} : <\delta, \sigma'>) \in A \Rightarrow \sigma \preceq \delta\ ) \\ &\quad \wedge (\ \neg\exists\ (\mathrm{v} : <\delta, \sigma>) \in A \Rightarrow \delta = f_{\mathrm{BaseType}}(\sigma)[*]\ ) \end{aligned}$$

$$\textsc{Cond} \quad : \quad \frac{A \vdash e : \textsc{bool} \quad A \vdash \mathit{Ass}_t;\ \mathit{Rest} : \tau \quad A \vdash \mathit{Ass}_e;\ \mathit{Rest} : \tau'}{A \vdash \textsc{if}\ (e)\ \{\mathit{Ass}_t;\}\ \textsc{else}\ \{\mathit{Ass}_e;\};\ \mathit{Rest} : \tau''}$$

$$\text{if } \exists\ \tau'' : \tau'' = f_{\mathrm{lub}}(\tau, \tau')$$

$$\textsc{Assert1} \quad : \quad \frac{A \vdash \mathrm{e} : \sigma}{A \vdash \textsc{assert}(\ \mathrm{e},\ \tau) : \tau} \qquad \text{if} \qquad \tau \preceq \sigma$$

$$\textsc{Assert2} \quad : \quad \frac{A \vdash \mathrm{e} : \sigma}{A \vdash \textsc{assert}(\ \mathrm{e},\ \tau) : \sigma} \qquad \text{if} \qquad \sigma \preceq \tau$$

Fig. 6. The basic type inference rules of SaC.

So far, only typing rules for function bodies have been presented. For typing function definitions and function applications the set of legal types has to be extended by function types. The type of an n-ary function that expects arguments of types $\tau_1, ..., \tau_n$ and returns values of type τ is denoted as $\bigotimes_{i=1}^{n} \tau_i \rightarrow \tau$.

In contrast to standard monomorphic type systems the typing rules for user-defined functions in SaC do not only serve to check a function's declared type against its body, but are also used to specialize shape-invariant functions for specific argument shapes. This can be elegantly formalized by making use of the subtyping relationship.

Figure 7 presents the type rules for user defined functions in SaC. The rules for typing function definitions, i.e. Prg, FunDef1 and FunDef2, allow more specific types to be inferred than the declared types. Since these specializations are a compiler introduced form of overloading, covariance has to be enforced in order to ensure type safety (see Castagna (1995) for an extended discussion of this topic). Which of the potentially infinite number of specializations actually is built is guided by the FunAp rule for typing function applications. It allows any argument type to be chosen which

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

$$\textsc{Prg} \quad : \quad \frac{\{\}\vdash \tau\ \textsc{main}()\ Body : \sigma}{\{\}\vdash FunDef_1,\ ...,\ FunDef_n\ \tau\ \textsc{main}()\ \{Body\} : \sigma}$$
$$\text{if } \sigma \preceq \tau$$

$$\textsc{FunDef1} \quad : \quad \frac{\{\}\vdash Body : \sigma}{\{\}\vdash \tau\ F()\ \{Body\} : \sigma}$$
$$\text{if } \sigma \preceq \tau$$

$$\textsc{FunDef2} \quad : \quad \frac{\{v_i : <\delta_i, \tau_i>\}\vdash Body : \tau}{A\vdash \delta F(\delta_1\ v_1,\ ...,\ \delta_n\ v_n)\ \{Body\} : \bigotimes_{i=1}^{n} \tau_i \rightarrow \tau}$$
$$\text{if } \tau_i \preceq \delta_i \wedge \tau \preceq \delta$$

$$\textsc{FunAp} \quad : \quad \frac{A\vdash e_i : \sigma_i \quad \{\}\vdash \delta F(\delta_1\ v_1,\ ...,\ \delta_n\ v_n)\ \{Body\} : \bigotimes_{i=1}^{n} \tau_i \rightarrow \tau}{A\vdash F(e_1 ... e_n) : \tau}$$
$$\text{if } \forall i \in \{1, ..., n\}\ :\ \sigma_i \preceq \tau_i \preceq \delta_i \wedge \tau \preceq \delta$$

Fig. 7. Specialization of user defined functions in SaC.

is a supertype of the actual argument type and a subtype of the declared parameter type. Although this ambiguity renders the type system non-unique, it can be shown that

- the type system is decidable, and that
- if two types α and β can be inferred for an expression e, either $\alpha \preceq \beta$ or $\beta \preceq \alpha$ holds.

Decidability results from the fact that all shape information can be neglected by introducing type assertions and falling back to the `[*]`-types which renders the type system isomorphic to standard monomorphic systems. The second property can be shown by induction over the typing rules.

Whereas user-defined functions can be specialized by analyzing the function bodies with assumptions about different argument shapes, built-in array operations, e.g. `dim` or `sel` have to be treated differently. For each of these functions, a type declaration similar to that of user defined functions is accompanied by an operation-specific type function CT, which computes from given argument types result types as specific as possible.[6]

The typing rule for applications of built-in array operations is given in the upper part of figure 8. It states that the type of an application of a primitive (built-in) operation F is computed by the type function CF_F associated to F, provided that the inferred argument types match the declared type of F. If the type function yields $\perp$, i.e. it detects a shape incompatibility, the function application under consideration can not be successfully typed.

[6] In fact, these type functions can be considered a special form of built-in dependent types.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

$$\textsc{PrfAp} \quad : \quad \frac{A\vdash\, e_i : \sigma_i \qquad A\vdash\, F : \bigotimes_{i=1}^{n} \tau_i \rightarrow \tau}{A\vdash\, F(e_1, ..., e_n) : \sigma}$$

$$\text{if } \ \forall i \in \{1, ..., n\} \ : \sigma_i \preceq \tau_i \wedge \mathrm{CT}_F(\sigma_1, ..., \sigma_n) = \sigma \neq \bot$$

where $\forall \alpha \in \{\ \text{INT, FLOAT, DOUBLE, CHAR, BOOL}\}$:

$\mathtt{dim}{::}\ \alpha[*] \rightarrow \text{INT}$

$\mathrm{CT}_{\mathtt{dim}}(x) = \text{INT}$

$\mathtt{sel}{::}\ \alpha[*] \times \text{INT}[\bullet] \rightarrow \alpha[*]$

$$\mathrm{CT}_{\mathtt{sel}}(x, y) = \begin{cases} \alpha[s_0, ..., s_{n-1}] & \text{iff } x = \alpha[s_0, ..., s_{m+n-1}] \wedge y = \text{INT}[m] \\ \alpha[\overbrace{\bullet, ..., \bullet}^{n}] & \text{iff } x = \alpha[\overbrace{\bullet, ..., \bullet}^{m+n}] \wedge y = \text{INT}[m] \\ \bot & \text{iff } x \preceq \alpha[\overbrace{\bullet, ..., \bullet}^{n}] \wedge y = \text{INT}[m] \wedge m > n \\ \alpha[*] & \text{otherwise} \end{cases}$$

Fig. 8. Typing primitive array operations in SaC.

As an example, the type declarations as well as the CT-functions of the built-in operations dim and sel are given in the lower part of figure 8. The function dim (cf. section 2.3), which computes an array's dimensionality, returns a scalar value irrespective of the shape of its argument. Therefore, the declared type ($\alpha[*] \rightarrow$ INT) of dim contains an exact shape on its right-hand side which renders the associated type function $\mathrm{CT}_{\mathtt{dim}}$ constant.

In contrast to dim, the selection function sel (cf. section 2.3) may return arrays of arbitrary shape which requires a (relatively imprecise) type declaration $\alpha[*] \times \text{INT}[\bullet] \rightarrow \alpha[*]$. The result shape of a given application of sel depends on the dimensionality of the argument array and the length of the selection vector. This relationship between argument and result types is captured by the associated type function $\mathrm{CT}_{\mathtt{sel}}$. If the shapes of both arguments are known and the length of the selection vector is smaller than or equal to the dimensionality of the array, an element/subarray is selected, i.e. the result shape is a postfix of the shape of the first argument. If only the dimensionality of the array is known, at least the dimensionality of the result can be determined. Should it turn out that the selection vector is longer than the number of dimensions of the array, then $\bot$ is returned by $\mathrm{CT}_{\mathtt{sel}}$, which results in a type error. In all other cases, the return type has to remain as general as the declared type.

The type inference for WITH-loops can be formalized in a similar way. Figure 9 shows the rules for typing GENARRAY-WITH-loops. The WlIdx rule is used to infer the length of the index-vector of a generator expression. The inference is done by means of a type function CT_{WlIdx} which computes the generator type from the types of the boundary, step, and width expressions. If at least one of them is an integer vector of some known length n and the others types are supertypes of INT[n], the generator can be successfully typed as INT[n]. It should be noted here that for vectors for which less specific types than INT[n] are inferred conformity checks have to be made at runtime. If no exact shapes are known for the four generator components, but their

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

$$\textsc{WlIdx} \quad : \quad \frac{\mathrm{A}\vdash e_1 : \tau_1 \quad \mathrm{A}\vdash e_2 : \tau_2 \quad \mathrm{A}\vdash e_3 : \tau_3 \quad \mathrm{A}\vdash e_4 : \tau_4}{\mathrm{A}\vdash \textsc{with}(e_1 <= iv <= e_2 \ \textsc{step}\ e_3 \ \textsc{width}\ e_4) : \tau}$$

$$\text{if } \mathrm{CT}_{WlIdx}(\tau_1, \tau_2, \tau_3, \tau_4) = \tau \neq \perp$$

$$\text{where } \mathrm{CT}_{WlIdx}(l, u, s, w) = \begin{cases} \textsc{int}[n] & \text{iff } \overset{1}{\exists}\, n : \textsc{int}[n] \preceq l, u, s, w \\ \textsc{int}[\bullet] & \text{iff } \textsc{int}[\bullet] \preceq l, u, s, w \\ \perp & \text{otherwise} \end{cases}$$

$$\textsc{WlGen} \quad : \quad \frac{\begin{array}{c} \mathrm{A}\vdash \textsc{with}(e_1 <= iv <= e_2 \ \textsc{step}\ e_3 \ \textsc{width}\ e_4) : \tau_{idx} \\ \mathrm{A}\{iv : <\tau_{idx}, \tau_{idx}>\}\vdash e_{val} : \tau_{val} \quad \mathrm{A}\vdash e_{shp} : \tau_{shp} \end{array}}{\mathrm{A}\vdash \textsc{with}(\ \dots iv \dots)\ \textsc{genarray}(\ e_{shp},\ e_{val}) : \sigma}$$

$$\text{if } \mathrm{CT}_{WlGen}(\tau_{idx}, \tau_{shp}, \tau_{val}) = \sigma \neq \perp$$

$$\text{where } \mathrm{CT}_{WlGen}(i, s, v) =$$

$$\begin{cases} \alpha[\overbrace{\bullet, \dots, \bullet}^{m+n}] & \text{iff } (\overset{1}{\exists}\, m : \textsc{int}[m] \preceq i, s) \ \wedge v \preceq \alpha[\overbrace{\bullet, \dots, \bullet}^{n}] \\ \perp & \text{iff } (\neg\exists\, m : \textsc{int}[m] \preceq i, s) \\ f_{\text{BaseType}}(v)[*] & \text{otherwise} \end{cases}$$

$$\textsc{WlGen2} \quad : \quad \frac{\begin{array}{c} \mathrm{A}\vdash \textsc{with}(e_1 <= iv <= e_2 \ \textsc{step}\ e_3 \ \textsc{width}\ e_4) : \tau_{idx} \\ \mathrm{A}\{iv : <\tau_{idx}, \tau_{idx}>\}\vdash e_{val} : \tau_{val} \end{array}}{\mathrm{A}\vdash \textsc{with}(\ \dots iv \dots)\ \textsc{genarray}(\ [s_0, \dots, s_{n-1}],\ e_{val}) : \sigma}$$

$$\text{if } \mathrm{CT}_{WlGen2}([s_0, \dots, s_{n-1}], \tau_{val}) = \sigma \ \wedge \textsc{int}[n] \preceq \tau_{idx}$$

$$\text{where } \mathrm{CT}_{WlGen2}([s_0, \dots, s_{n-1}], v) =$$

$$\begin{cases} \alpha[s_0, \dots, s_{m-1}] & \text{iff } v = \alpha[s_n, \dots, s_{m-1}] \\ \alpha[*] & \text{otherwise} \end{cases}$$

Fig. 9. Typing With-loops in SaC.

types do include integer vectors, the generator under consideration is typed INT[•]. Again, this requires conformity checks at runtime. For all other type combinations a type error is produced.

The rule WLGen specifies by means of yet another type function CT_{WlGen} how the type of a genarray-WITH-loop is computed from the type of its generator part, the type of its shape expression, and the type of its value expression. Only if the lengths of the shape expression and the generator expressions may be the same, i.e. there exists an *m* so that INT[m] is a subtype of the shape expression type and the generator type, the genarray-WITH-loop can be successfully typed. Furthermore, a more specific type than $\alpha[*]$, where α is the element type of the value expression, can only be inferred, if (i) the length of the shape or the generator expression is statically known, and if (ii) at least the dimensionality of the value expression can be determined. If so, the dimensionality of the genarray-WITH-loop can be inferred as the length of the shape expression plus the dimensionality of the value expression.

A second rule for genarray-WITH-loops, WLGen2, allows to infer exact shapes for genarray-WITH-loops with constant shape expression. The significant difference to WLGen is that the result shape is computed from the shape expression itself rather than from its type. Therefore, it is applicable to constant shape expressions only.

Similar rules can be formulated for MODARRAY- and FOLD-WITH-loops.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

4 Compilation of SaC programs

This section focuses on compilation issues, particularly on measures taken in the actual SaC compiler release[7] to achieve runtime performances comparable to that of low-level programs.

First, it has to be guaranteed that individual array operations can be compiled into code which can be executed efficiently. Most array operations in SaC being specified as WITH-loops, it primarily suffices to find a suitable compilation scheme that generates code for these constructs. However, dealing with statically unknown dimensionalities requires a rather complex index generation scheme which introduces quite some overhead when compared to static loop nestings. To avoid this overhead, the actual SaC compiler infers the dimensionalities of the arrays to which shape-invariant operations are actually applied. This is done as part of the type inference process which exploits the latitude of the type system to specialize functions wrt. arbitrary subtypes of their formal parameter types.

Whenever this process yields exact shapes rather than just dimensionalities, the code can be further improved by exploiting the map-like definition of WITH-loops. Since the order of individual WITH-loop indices can be arbitrarily permuted without affecting the overall result, the loop nestings can be arranged so that the order of array accesses at runtime is adjusted to the cache characteristics of the intended target architecture. This property facilitates the implementation of several optimization techniques to this effect such as *Loop Tiling* and *Array Padding* (Lam *et al*., 1991; Wolf & Lam, 1991b; Coleman & McKinley, 1995; Manjikian & Abdelrahman, 1995) as no data dependencies or loop alignments have to be taken care of. Details on the incorporation of these optimizations into the SaC compiler can be found in Grelck (2001).

However, being able to compile individual array operations (WITH-loops) into efficiently executable code only constitutes the first step of compiling APL-like specifications as introduced in section 2. More challenging efficiency problems arise from the compilation of nested applications of such operators. A naïve compilation of such expressions introduces intermediate arrays, which degrades performance due to higher rates of memory accesses and increased memory consumption.

Many sophisticated compilation techniques for optimizing conventional loop nestings and arrays have been developed (for surveys, see Zima & Chapman (1991), Gao *et al.* (1993), Wolfe (1995), Lewis *et al.* (1998) and Allen & Kennedy (2001)). Most of these techniques are based on data dependency analysis (Allen & Kennedy, 1987; Wolf & Lam, 1991a; Yi *et al*., 2000) which, for loops in general, provides the essential criteria to decide what kind of loop transformations can be safely applied.

On the level of WITH-loop expressions in SaC, the situation is rather different as the language design guarantees several important properties. Each WITH-loop represents a multi-dimensional loop nesting that operates on the same domain of data. In most cases, these domains are even known precisely at compile time. Per definition, there are no side-effects or data dependencies between different instances

[7] See <http://www.informatik.uni-kiel.de/~sacbase/>.

of such loop nestings. These properties allow for more radical optimizations as the loop instances can be arbitrarily permuted and the computation of individual array elements can be deferred by forward substitution without effecting the overall result. This observation leads to an optimization on the level of WITH-loops, called *With Loop Folding*, which systematically eliminates intermediate arrays.

Since most array operations in SAC are defined by means of WITH-loops, this single optimization technique suffices to eliminate intermediate arrays between arbitrary array operations.

Formally, *With Loop Folding* is based on the well known map equation

$$\texttt{map f} \circ \texttt{map g} \equiv \texttt{map (f} \circ \texttt{g)}$$

but has been extended to handle restricted (index) domains as they result from structural operations where the generator parts do not cover the entire index range of the array to be created/modified. To exploit the full potential of this optimization, static knowledge of the exact generator sets and the array shapes involved is essential. Therefore, function specialization during type inference is enforced to exact argument shapes as far as possible. Furthermore, the entire range of standard optimizations, in particular *Constant Folding* and *Common Subexpression Elimination* have to be applied prior to *With Loop Folding* in order to increase the number of generator boundaries whose values are statically available.

There are also memory related issues to be considered to achieve competitive runtimes. With arrays that consume significant fractions of the available memory, it is essential to destructively update arrays whenever possible, without violating referential transparency. In particular with sequences of `modarray` operations, of which each changes a few elements only, a single superfluous intermediate array can spoil the entire runtime performance. Therefore, garbage collection schemes that postpone the identification of garbage until a pre-specified amount of memory has been used up, e.g. *Mark-Sweep Collection* (Cohen, 1981) based schemes or *Copying Collection* (Cohen, 1981) based schemes, are not suitable for SAC. Instead, *Reference Counting* (Cohen, 1981) is used which, at runtime, identifies and removes garbage as soon as the last access to it has been made. Similar to the approach taken in SISAL, referential transparency allows for several code reordering optimizations to avoid superfluous reference counting operations and to maximize potential memory reuse (Cann, 1989).

However, the drawback of the reference counting approach is that irrespective of the size of the data structure it is applied to, a fixed amount of administrative overhead is required. Although this overhead can usually be neglected for operations on large arrays, for small data structures this is not the case. In particular, when taking into account that in SAC all data structures – including scalars, index vectors, and other small arrays – conceptually are arrays, a uniform application of reference counting to all arrays is not feasible. Instead, the compilation process, again, makes use of the static availability of shape information: scalars, i.e. arrays of dimensionality zero, and arrays with only a few elements are allocated on the runtime stack rather than on the heap. This does not only avoid overhead due to

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

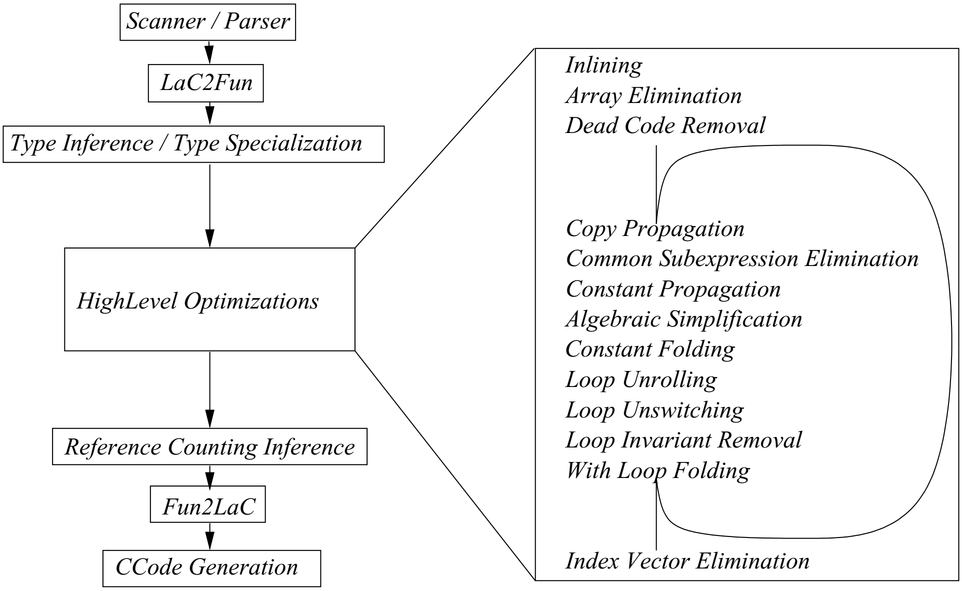


Fig. 10. Compiling SaC programs into C programs.

reference counting, but also indirect accesses as they are required for heap allocated objects.

After giving a brief outline of the compilation process, the following subsections focus on *With Loop Folding* and on the compilation of WITH-loops to static loop nestings.

4.1 An outline of the compilation process

The natural choice of a target language for the compilation of SaC is C. Compilation to C can be liberated from all hardware-specific low-level optimizations such as delay-slot utilization or register allocation, as this is taken care of by the C compiler for the target machine. Last not least, the strong syntactical similarity between the two languages allows the compilation efforts to be concentrated on adequate array representations and on optimizing the code for array operations. Other basic language constructs can be translated more or less one to one to their C counterparts.

The major phases of the actual SaC compiler are shown in figure 10. After scanning and parsing the SaC-program to be compiled, its internal representation is simplified by a transformation called *LaC2Fun* which eliminates syntactical sugar such as loop constructs and (non-top-level) conditionals.

The next compilation phase implements a type inference algorithm based on the type rules described in the preceding section. To achieve utmost code optimizations, the actual implementation tries to specialize all array types to specific shapes. Starting from the designated `main` function, it traverses function bodies from outermost to innermost, propagating exact shapes as far as possible. To avoid non-termination, the number of potential function specializations is limited by a pre-specified number

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

of instances. If this number is exceeded, the generic version is used instead. However, since for most application programs all shapes can be statically inferred, the actual compiler implementation is restricted to such programs, i.e. it is assumed that after type inference, all shapes are statically known.

The fourth compilation phase implements all the optimizations that can be done on the level of SaC itself.[8] Of particular interest in this context are three 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 folding consecutive WITH-loops into single ones. It constitutes the key optimization for achieving competitive runtimes and is therefore discussed in some detail in the next subsection.
- *Array Elimination* replaces arrays with with less than a pre-specified number of elements in their data vectors by sets of SaC scalars which in turn are implemented as scalars in the generated C code.
- *Index Vector Elimination* tries to replace by scalar offsets into data vectors integer vectors used for indexing, anticipating row-major order.

To improve the applicability of these optimizations, constants have to be propagated/inferred as far as possible, i.e. several standard optimizations have to be included in this compilation phase as well. It also turns out that on the SaC level these standard optimizations, due to the absence of side-effects, can be applied much more rigorously than in state-of-the-art C compilers. The standard optimizations implemented in the actual compiler include Function Inlining, Constant Folding, Constant Propagation, Dead Code Removal, etc. (cf. figure 10).

Many of these optimizations interact with each other, e.g., constant folding may enable *With Loop Folding* by inferring exact generator boundaries of WITH-loops which, in turn, may enable further constant folding within the body of the resulting WITH-loop. Therefore, the optimizations are applied in a cyclic fashion, as shown on the right hand side of figure 10. This cycle terminates if either there are no more code changes or if a pre-specified number of cycles has been performed.

The three final compilation phases transform the optimized SaC code step by step into a C program. The first phase, called *Reference Counting Inference*, adds for all non-scalar arrays operations that handle the reference counters at runtime. The techniques used here are similar to those developed for Sisal.

The next phase, called *Fun2LaC*, is dual to *LaC2Fun*; it reverts tail-end recursive functions into loops and inlines functions that were created from non-top-level conditionals during *LaC2Fun*.

Finally, the SaC-specific language constructs are compiled into ANSI C code. The most interesting aspect of this phase is the code generation for WITH-loops, which can be parameterized by the cache characteristics of the target architecture. It is described in some more detail in section 4.3.

[8] It should be noted here, that in fact some slight extensions of SaC to be mentioned in the next subsections are required.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

4.2 *With Loop Folding*

As explained earlier, the key idea of *With Loop Folding* is to provide a scheme that replaces functional compositions of compound array operations by a single array operation which realizes the functional composition without creating intermediate arrays.

The simplest of these transformations is the composition of two WITH-loops which maps functions f and g to all elements of an array. Assuming A to be an array with element type τ and f and g to be functions of type $\tau \rightarrow \tau$, then

```
{...
  B = with( . <= iv <= . )
      modarray( A, iv, f( A[iv] ));
  C = with( . <= jv <= . )
      modarray( B, jv, g( B[jv] ));
...}
```

can be replaced by

```
{...
  C = with( . <= jv <= . )
      modarray( A, jv, g( f( A[jv] ) ));
...}
```

provided that B is not referenced anywhere else in the program.

In this restricted setting the intended optimization directly corresponds to the well known map equation map f $\circ$ map g $\equiv$ map (f $\circ$ g), which is fundamental to several optimizations such as *Deforestation* (Wadler, 1990; Chin, 1994; Gill, 1996; Nemeth & Peyton Jones, 1998) in the context of operations on lists and *Loop Fusion* (Bacon *et al*., 1994; Zima & Chapman, 1991; Wolfe, 1995; Allen & Kennedy, 2001) in the context of conventional (sequential) loops.

However, in a more general setting

- the WITH-loops to be folded may have non-identical index sets in their generator parts;
- the second WITH-loop may contain several references to the elements of the array defined by the first one;
- the access(es) to the array defined by the first WITH-loop may be non-local, i.e. instead of B[jv] expressions of the form B[I_op(jv)] are allowed where I_op projects index vectors to index vectors.

The piece of SAC program given in the upper part of figure 11 highlights these features in a nutshell, and will therefore be used as a running example throughout this section. It consists of two WITH-loops which successively compute vectors B and C from a given vector A. Each of these vectors consists of 80 integer numbers. The first WITH-loop defines B to differ from A in that the first 40 elements are incremented by 3, the second WITH-loop defines C to differ from B in that the last 60 elements of C are computed as the sum of two elements of B, the actual one and the one that is located at the actual index position minus 10.

These WITH-loops are graphically depicted in the lower part of figure 11: each horizontal bar shows all elements of the vector named to the left of it. The index

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

```
{...
  B = with( [0] <= iv < [40])
      modarray( A, iv, A[iv] + 3);
  C = with( [20] <= jv < [80])
      modarray( B, jv, B[jv] + B[jv - [10]]);
... }
```

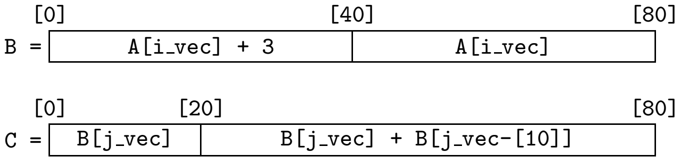


Fig. 11. Two successive WITH-loops with overlaping index ranges and multiple references.

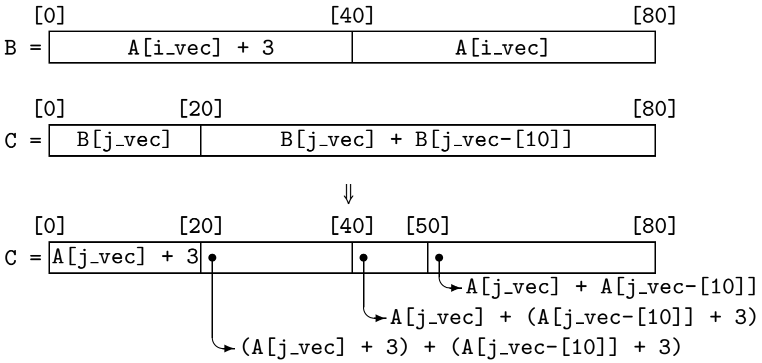


Fig. 12. Substituting two successive array modifications by a single one.

vectors on top of the bars indicate the positions of the respective elements within the bars. The SAC expressions inscribed in the bars define how the vector elements are computed from the elements of other vectors. Since different computations are required in different index vector ranges the bars are accordingly divided up by vertical lines.

Instead of first computing B from A then C from B, the array C can be computed from the array A directly. As depicted in figure 12, this operation requires four index ranges of C to be treated differently, which cannot be expressed by a single WITH-loop anymore. To remedy this problem, a more general version of WITH-loops needs to be introduced which is not part of SAC proper, but only internally used by the SAC compiler. It allows an arbitrary number of generator operation pairs to be specified, provided that (i) the index vector sets defined by the generators are disjoint, and that (ii) their union is the set of all legal index vectors of the result array.[9]

For these internal WITH-loops the following notation will be used:

[9] (ii) is required for genarray- and modarray-WITH-loops only.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

For $i \in \{1, ..., m\}$ let ($\mathtt{l}_i$ <= iv <= $\mathtt{u}_i$ step $\mathtt{s}_i$ width $\mathtt{w}_i$) be legal generator expressions that denote disjoint index vector sets, and let $\mathtt{Op}_i$(iv) be expressions that evaluate to arrays of type $\tau[s_0, ..., s_{l-1}]$ for all index vectors iv from the respective index vector sets. Then

```
A = with
      ( $l_1$ <= iv <= $u_1$ step $s_1$ width $w_1$) :$Op_1$( iv)
        ⋮                                           ⋮
      ( $l_m$ <= iv <= $u_m$ step $s_m$ width $w_m$):$Op_m$( iv)
    genarray( [$r_0$, ..., $r_{k-1}$]);
```

defines an array A of shape [$\mathtt{r}_0$, ..., $\mathtt{r}_{k-1}$, $\mathtt{s}_0$, ..., $\mathtt{s}_{l-1}$] with element type τ where A[iv] := $\mathtt{Op}_j$(iv) $\Leftrightarrow$ iv $\in \mathscr{Gen}(\mathtt{l}_j, \mathtt{u}_j, \mathtt{s}_j, \mathtt{w}_j)$, provided that

(i) $\forall_{i,j \in \{1...m\}} : \mathscr{Gen}(\mathrm{l}_i, \mathrm{u}_i, \mathrm{s}_i, \mathrm{w}_i) \cap \mathscr{Gen}(\mathrm{l}_j, \mathrm{u}_j, \mathrm{s}_j, \mathrm{w}_j) \neq \emptyset \Rightarrow (i = j)$, and

(ii) $\bigcup_{j=1}^{m} \mathscr{Gen}(\mathrm{l}_j, \mathrm{u}_j, \mathrm{s}_j, \mathrm{w}_j) = \mathscr{Liv}([r_0, ..., r_{k-1}])$;

otherwise, it is undefined.

Note that this notation serves as a generalized form of genarray- and modarray-WITH-loops. fold-WITH-loops have a slightly different internal representation which uses an expression of the form fold(*fun*, *neutr*) instead of genarray([$\mathtt{r}_0$, ..., $\mathtt{r}_{k-1}$]).

With this notation at hand, our example problem can be specified as follows: Find a transformation scheme which transforms

```
B = with
      ( [ 0] <= iv < [40] ):A[iv] + 3
      ( [40] <= iv < [80] ):A[iv]
    genarray( [80]);

C = with
      ( [ 0] <= jv < [20] ):B[jv]
      ( [20] <= jv < [80] ):B[jv] + B[jv - [10]]
    genarray( [80]);
```

into

```
C = with
      ( [ 0] <= jv < [20] ):A[jv] + 3
      ( [20] <= jv < [40] ):(A[jv] + 3) + (A[jv - [10]] + 3)
      ( [40] <= jv < [50] ):A[jv] + (A[jv - [10]] + 3)
      ( [50] <= jv < [80] ):A[jv] + A[jv - [10]]
    genarray( [80]);
```

The basic idea is to define a scheme which takes two internal WITH-loops, and step by step replaces all references to the result of the first WITH-loop (the array B in our example) by their definitions. Once all references to that array are replaced, its defining WITH-loop can be eliminated.

Figure 13 gives the rule for a single replacement step. The upper part shows the most general pattern for an application of the replacement rule: a function body contains two WITH-loops; the first WITH-loop defines an array A whose elements are

```
{ ...
  A = with
         IV_{1,1}( iv) :Op_{1,1}( iv)
            ⋮             ⋮
         IV_{1,m}( iv):Op_{1,m}( iv)
      genarray( shp);

  ...
  B = with
         IV_{2,1}( jv):Op_{2,1}( jv)
            ⋮             ⋮
         IV_{2,i}( jv) :Op_{2,i}( jv)  = ⊣...A[ I_op( jv)]...⊢
            ⋮             ⋮
         IV_{2,n}( jv):Op_{2,n}( jv)
      wl_oper;

  ...}

                              ⇓

{ ...
  A = with
         IV_{1,1}( iv) :Op_{1,1}( iv)
            ⋮             ⋮
         IV_{1,m}( iv):Op_{1,m}( iv)
      genarray( shp);

  ...
  B = with
         IV_{2,1}( jv)  :Op_{2,1}( jv)
            ⋮               ⋮
         IV_{2,i,1}( jv):Op_{2,i,1}( jv)  = ⊣...Op_{1,1}( I_op( jv))...⊢
            ⋮               ⋮
         IV_{2,i,m}( jv):Op_{2,i,m}( jv)  = ⊣...Op_{1,m}( I_op( jv))...⊢
            ⋮               ⋮
         IV_{2,n}( jv)  :Op_{2,n}( jv)
      wl_oper;

  ...}
```

$$\text{with } \mathtt{IV}_{2,i,1} := \{\ \mathtt{jv}\ |\ \mathtt{jv} \in \mathtt{IV}_{2,i} \wedge \mathtt{I\_op(\ jv)} \in \mathtt{IV}_{1,1}\}$$

$$\vdots \qquad \vdots$$

$$\mathtt{IV}_{2,i,m} := \{\ \mathtt{jv}\ |\ \mathtt{jv} \in \mathtt{IV}_{2,i} \wedge \mathtt{I\_op(\ jv)} \in \mathtt{IV}_{1,m}\}$$

Fig. 13. Single WITH-loop-folding step.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

referred to in at least one expression $\texttt{Op}_{2,i}\texttt{(jv)}$ of the second WITH-loop, as indicated by $\dashv\texttt{...A[I\_op(jv)]...}\vdash$. For reasons of convenience, all generator expressions are denoted by $\texttt{IV}_{idx}\texttt{(iv)}$ which is considered a notational shortcut for (l_{idx} <= iv <= u_{idx} step S_{idx} width W_{idx}). When referring to the set of index vectors defined by $\texttt{IV}_{idx}\texttt{(iv)}$, i.e. $\mathcal{G}en(\mathrm{l}_{idx}, \mathrm{u}_{idx}, \mathrm{s}_{idx}, \mathrm{w}_{idx})$, $\texttt{IV}_{idx}$ is used without the trailing index vector name in brackets. *wl_oper* at the end of the second WITH-loop indicates that the replacement rule can be applied irrespective of the internal WITH-loop version actually used.

To replace A[I_op(jv)] in $\texttt{Op}_{2,i}\texttt{(jv)}$ by its definition, it has to be determined to which index vector set from the definition of A the projected index vectors I_op(jv) belong. Since they may refer to more than one of these sets, in general, $\texttt{IV}_{2,i}$ has to be split up into m subsets $\texttt{IV}_{2,i,1}$, ..., $\texttt{IV}_{2,i,m}$, each of which contains those elements of $\texttt{IV}_{2,i}$ whose mappings wrt. I_op are in the index vector sets $\texttt{IV}_{1,1}$, ..., $\texttt{IV}_{1,m}$, respectively. The expressions associated to these sets are derived from $\texttt{Op}_{2,i}\texttt{(jv)}$ by replacing A[I_op(jv)] with the respective definitions $\texttt{Op}_{1,1}\texttt{(I\_op(jv))}$, ..., $\texttt{Op}_{1,m}\texttt{(I\_op(jv))}$, i.e.

$$\texttt{IV}_{2,i}\texttt{(jv)} = \dashv\texttt{...A[I\_op(jv)]...}\vdash$$

from the definition of the second WITH-loop is replaced with

$$\begin{array}{lcl} \texttt{IV}_{2,i,1}\texttt{(jv)} & : & \texttt{Op}_{2,i,1} = \dashv\texttt{... Op}_{1,1}\texttt{(I\_op(jv))...}\vdash \\ \vdots & \vdots & \quad\vdots \\ \texttt{IV}_{2,i,m}\texttt{(jv)} & : & \texttt{Op}_{2,i,1} = \dashv\texttt{... Op}_{1,m}\texttt{(I\_op(jv))...}\vdash \quad . \end{array}$$

However, this rule can only be applied if it can be made sure that such generators $\texttt{IV}_{2,i,j}\texttt{(jv)}$ indeed exist, i.e. if subsets of the form $\{$ jv | jv $\in \mathrm{IV}_{2,i} \wedge$ I_op(jv) $\in \mathrm{IV}_{1,1}\}$ always can be denoted by generator expressions. After restricting I_op to linear transformations, this task eventually boils down to computing intersections of generator-defined index vector sets. It can be assumed without loss of generality that the widths of the generators under consideration are 1 in all dimensions, since other generators with width components greater than 1 can be split up into several element-wise shifted generators of widths 1.

Given two such generators $\mathcal{G}en(\mathrm{l}_A, \mathrm{u}_A, \mathrm{s}_A, [1...1])$ and $\mathcal{G}en(\mathrm{l}_B, \mathrm{u}_B, \mathrm{s}_B, [1...1])$, their intersection indeed can be denoted by a generator expression. The key observation to be made here is that whenever there exist two index vectors iv and iv′ from $\mathcal{G}en(\mathrm{l}_A, \mathrm{u}_A, \mathrm{s}_A, [1...1]) \cap \mathcal{G}en(\mathrm{l}_B, \mathrm{u}_B, \mathrm{s}_B, [1...1])$, their difference is a multiple of s_A and of s_B. As a consequence, their difference is also a multiple of the least common multiple of s_A and s_B which thus can be used as step vector of the intersection. The generator boundaries of the intersection can be computed from the element-wise maxima and minima of the lower and upper bounds, respectively. This value might have to be adjusted only for the lower bound since the maximum of the lower bounds is not necessarily an element of the intersection if the generators are not dense ($\mathrm{s}_a \neq [1...1] \wedge \mathrm{s}_B \neq [1...1]$). Therefore, within the first period an intersecting index vector has to be looked for. If found, it serves as lower bound, otherwise, the

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

```
B = with
      ( [ 0] <= iv < [40] ):A[iv] + 3
      ( [40] <= iv < [80] ):A[iv]

C = with                                                              (a)
      ( [ 0] <= jv < [20] ):B[jv]
      ( [20] <= jv < [80] ):B[jv] + B[jv - [10]]

                                  ⇓

C = with                                                              (b)
      ( [ 0] <= jv < [20] ):A[jv] + 3
      ( [20] <= jv < [80] ):B[jv] + B[jv - [10]]

                                  ⇓

C = with                                                              (c)
      ( [ 0] <= jv < [20] ):A[jv] + 3
      ( [20] <= jv < [40] ):(A[jv] + 3) + B[jv - [10]]
      ( [40] <= jv < [80] ):A[jv] + B[jv - [10]]

                                  ⇓

C = with                                                              (d)
      ( [ 0] <= jv < [20] ):A[jv] + 3
      ( [20] <= jv < [40] ):(A[jv] + 3) + (A[jv - [10]] + 3)
      ( [40] <= jv < [80] ):A[jv] + B[jv - [10]]

                                  ⇓

C = with                                                              (e)
      ( [ 0] <= jv < [20] ):A[jv] + 3
      ( [20] <= jv < [40] ):(A[jv] + 3) + (A[jv - [10]] + 3)
      ( [40] <= jv < [50] ):A[jv] + (A[jv - [10]] + 3)
      ( [50] <= jv < [80] ):A[jv] + A[jv - [10]]
```

Fig. 14. Stepwise WITH-loop-folding at the example presented in figure 12.

intersection is empty. More formally, we have:

$$\mathscr{G}en(\mathrm{l}_A,\ \mathrm{u}_A,\ \mathrm{s}_A,\ [1...1])\ \cap\ \mathscr{G}en(\mathrm{l}_B,\ \mathrm{u}_B,\ \mathrm{s}_B,\ [1...1])$$

$$= \begin{cases} \mathscr{G}en(\mathrm{l},\ \mathrm{u},\ \mathrm{s},\ [1...1]) & \text{iff } \exists\ x, y \in \mathbb{N}_0^n : \mathrm{l}_{min} \leqslant \mathrm{l}_A + x * \mathrm{s}_A = \mathrm{l}_B + y * \mathrm{s}_B \leqslant \mathrm{l}_{max} \\ \varnothing & \text{otherwise} \end{cases}$$

where

$$\begin{array}{ll} \mathrm{s} = \mathrm{lcm}(\mathrm{s}_A, \mathrm{s}_B) & \mathrm{u} = \min(\mathrm{u}_A, \mathrm{u}_B) \\ \mathrm{l}_{min} = \max(\mathrm{l}_A, \mathrm{l}_B) & \mathrm{l}_{max} = \min(\mathrm{l}_{min} + \mathrm{s} - 1, \mathrm{u}) \\ \mathrm{l} = \mathrm{l}_A + x * \mathrm{s}_A & \end{array}$$

where lcm computes the least common multiple and all operations are considered to be applied element-wise to the vector arguments given.

Applying the above replacement rule to the example problem we get a sequence of program transformations as shown in figure 14. Starting out from the internal representation of the two given WITH-loops (figure 14(a)), it shows the stepwise

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

transformation of the second WITH-loop construct until the final version which does not contain any references to the array B anymore in figure 14(e) is reached. Each of these steps results from a single application of the *With Loop Folding* rule of figure 13; the references to B which are replaced in the next transformation step in each of the intermediate forms figure 14(a) to figure 14(d) are marked by boxes.

4.3 Code Generation for WITH-loops

The definition of multi-generator WITH-loops as introduced in the previous subsection allows the order in which the index vector sets are traversed to be arbitrarily permuted without affecting the overall result. This property can be exploited when it comes to compiling them into C code. The simplest approach to do so is to compile each generator into a separate nesting of FOR-loops whose innermost body contains the associated expression to be evaluated. A naïve compilation scheme for this approach is presented in figure 15. It consists of rules of the form $\mathcal{C}[\![expr]\!] = expr'$ which denote context-free substitutions of SAC program fragments *expr* by C program fragments *expr'*. The rules apply only to multi generator WITH-loops. A compilation scheme for entire SAC programs is beyond the scope of this paper and would not provide any further insights into the code generation for WITH-loops.

Rules (1) and (2) apply to the two variants of multi generator WITH-loops. They differ in two respects: the initialization of the variable that holds the result and the recursive application of the compilation scheme to the individual generators.

The GENARRAY-variant allocates the result array using `a = MALLOC(` *shp*`)`. An explicit initialization is not required as the generator sets are guaranteed to be a partition of all legal index vectors. In the applications of the compilation scheme to the individual generators, the expressions to be evaluated are transformed into assignments of the form `a[iv] = Op( iv)`, which ensures correct insertion of the computed values into the result array.

In contrast, the compiled code for FOLD-WITH-loops starts out with an initialization of the result by the neutral element. The actual fold operation is generated in the course of compiling the generators, which is triggered by an application of the compilation scheme to generators that have been modified accordingly.

The last two rules concern the compilation of generator expressions into nestings of FOR-loops. As shown in rule (3), for each component of the indexing vector `iv` two nested FOR-loops are created: an outer loop for initializing and stepwise increasing the appropriate index vector component, and an inner loop for treating width components larger than `1`. The body of the inner loop derives from recursively applying the compilation scheme to the generator with its leading index vector components being eliminated. The creation of the innermost loop body is described by rule (4). It simply replaces the empty generator by the assignment associated with it.

Unfortunately, naïve compilation has two major problems. First, separately compiling the generators often introduces a considerable amount of loop overhead. One source are adjacent generators that perform identical operations. Another source are non-dense generators that have almost identical boundaries. They lead to separate

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} \texttt{a} = \texttt{with} \\ \qquad \texttt{(l}_1\texttt{ <= iv <= u}_1\texttt{ step s}_1\texttt{ width w}_1\texttt{)} \;\; : \texttt{Op}_1\texttt{(iv)} \\ \qquad\quad \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad \texttt{(l}_m\texttt{ <= iv <= u}_m\texttt{ step s}_m\texttt{ width w}_m\texttt{)} : \texttt{Op}_m\texttt{(iv)} \\ \quad \texttt{genarray(} \; shp\texttt{);} \end{array}\right]\!\!\right] \tag{1}$$

$$= \left\{\begin{array}{l} \texttt{a = MALLOC(} \; shp\texttt{);} \\ \texttt{iv = MALLOC(shape(} \; l_1\texttt{));} \\ \mathcal{C}[\![\texttt{(l}_1\texttt{ <= iv <= u}_1\texttt{ step s}_1\texttt{ width w}_1\texttt{) : a[iv] = Op}_1\texttt{(iv)}]\!] \\ \vdots \\ \mathcal{C}[\![\texttt{(l}_m\texttt{ <= iv <= u}_m\texttt{ step s}_m\texttt{ width w}_m\texttt{) : a[iv] = Op}_m\texttt{(iv)}]\!] \end{array}\right.$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} \texttt{a} = \texttt{with} \\ \qquad \texttt{(l}_1\texttt{ <= iv <= u}_1\texttt{ step s}_1\texttt{ width w}_1\texttt{)} \;\; : \texttt{Op}_1\texttt{(iv)} \\ \qquad\quad \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad \texttt{(l}_m\texttt{ <= iv <= u}_m\texttt{ step s}_m\texttt{ width w}_m\texttt{)} : \texttt{Op}_m\texttt{(iv)} \\ \quad \texttt{fold(} \; fun, \; neutr\texttt{);} \end{array}\right]\!\!\right] \tag{2}$$

$$= \left\{\begin{array}{l} \texttt{a = } neutr\texttt{;} \\ \texttt{iv = MALLOC(shape(} \; l_1\texttt{));} \\ \mathcal{C}[\![\texttt{(l}_1\texttt{ <= iv <= u}_1\texttt{ step s}_1\texttt{ width w}_1\texttt{) : a = } fun\texttt{(a, Op}_1\texttt{(iv))}]\!] \\ \vdots \\ \mathcal{C}[\![\texttt{(l}_m\texttt{ <= iv <= u}_m\texttt{ step s}_m\texttt{ width w}_m\texttt{) : a = } fun\texttt{(a, Op}_m\texttt{(iv))}]\!] \end{array}\right.$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} \texttt{([}l_i ... l_{n-1}\texttt{] <= iv <= [}u_i ... u_{n-1}\texttt{]} \\ \qquad \texttt{step [}s_i ... s_{n-1}\texttt{] width [}w_i ... w_{n-1}\texttt{]) : Ass} \end{array}\right]\!\!\right] \tag{3}$$

$$= \left\{\begin{array}{l} \texttt{for(iv[}i\texttt{] = }l_i\texttt{; iv[}i\texttt{] <= }l_i\texttt{; iv[}i\texttt{] += }s_i\texttt{-}w_i\texttt{) \{} \\ \quad \texttt{stop = MIN(iv[}i\texttt{]+}w_i\texttt{-1, }l_i\texttt{);} \\ \quad \texttt{for(; iv[}i\texttt{]<=stop; iv[}i\texttt{]++) \{} \\ \qquad \mathcal{C}\left[\!\!\left[\begin{array}{l} \texttt{([}l_{i+1} ... l_{n-1}\texttt{] <= iv <= [}u_{i+1} ... u_{n-1}\texttt{]} \\ \qquad \texttt{step [}s_{i+1} ... s_{n-1}\texttt{] width [1...1]) : Ass} \end{array}\right]\!\!\right] \\ \quad \texttt{\}} \\ \texttt{\}} \end{array}\right.$$

$$\mathcal{C}[\![\texttt{([] <= iv <= [] step [] width []) : Ass }]\!] = \texttt{Ass;} \tag{4}$$

Fig. 15. Compilation of multi generator WITH-loops.

loop nestings which could be reused if the non-dense generators were merged properly.

The second problem of naïve compilation results from the intricacies of the executing machinery, in particular from data caching. Whereas the organization of caches by lines favors memory accesses to adjacent addresses (so-called *spatial reuse* (Hennessy & Patterson, 1995)), the memory access patterns that result from naïve compilation – in general – turn out to be rather ragged.

Although elaborate C compilers provide several optimizations for rearranging loops, e.g. loop-fusion, loop-splitting and loop-permutation (Wolfe, 1995; Zima & Chapman, 1991; Allen & Kennedy, 2001), they often fail to significantly improve naïvely compiled code. The major problem these compilers have to deal with is the lack of information concerning the special form of FOR-loops as they are created by naïve compilation. C compilers do not statically know that all innermost loop bodies can be computed independently. Instead, they have to analyze the loop nestings for potential data dependencies which in C may be hidden in side-effecting function calls. Furthermore, C compilers cannot easily detect that all loop nestings generated from a single WITH-loop, together, assign a value to each index position in the result array exactly once. They have to apply optimization schemes based on cost heuristics that identify loop nestings for which a loop rearrangement most likely will improve spatial reuse (Ding, 2000).

These observations lead to the idea of systematically transforming naïvely compiled code into loop nestings that obey a specific order, the so called *canonical order* (Grelck *et al.*, 2000). Computing the array elements in canonical order means that the addresses of the resulting array elements are sorted in strictly ascending order irrespective of the form the generators involved have. This guarantees good spatial reuse for the write accesses to the resulting array and in many applications leads to good spatial reuse of the read accesses as well. Furthermore, it reduces the loop overhead by merging loop nestings whose index ranges overlap in outer dimensions.

Due to the variety of loop nestings that result from naïve compilation, such a transformation scheme in its general form requires rather complex loop modifications. However, the basic principle of this transformation can be exposed in a rather restricted setting. For WITH-loops that consist of dense generators only, i.e. all step components and all width components are 1, the loop nestings generated by naïve compilation consist of loops of the form

```
for( iv[i]=l; iv[i]<=u; iv[i]+=0){
  stop = MIN( iv[i] +1-1, u);
  for( ; iv[i]<=stop; iv[i]++){
   Body;
  }
}
```

only, which can be simplified to

```
for( iv[i]=l; iv[i]<=u; iv[i]++){
   Body;
}
```

For such loop nestings simple loop splitting and loop fusion operations suffice to establish canonical order. An example to this effect is shown in figure 16. It consists of a multi generator WITH-loop that generates a two-dimensional array a by means of three dense generators. While the first two generators define how to compute the elements of the upper and the lower left segment of the array, the third generator defines how to compute the remaining right half of the array.

Applying naïve compilation, loop nestings as shown in the middle section of figure 16 are created. They compute all elements of the first generator (lines (3)–(7)), before computing any elements of the third generator (lines (13)–(17)). Assuming that all arrays are stored in row-major order, this violates the canonical order which requires the elements of a to be computed one row after the other.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

```
a = with
      ( [0,0] <= iv <= [4,4]) : OP1;
      ( [5,0] <= iv <= [9,4]) : OP2;
      ( [0,5] <= iv <= [9,9]) : OP3;
    genarray( [10,10]);
```

⇓Naïve Compilation

```
a = MALLOC( [10,10]);                                   ( 1)
iv = MALLOC( [2]);                                      ( 2)
for( iv[0]=0; iv[0]<=5; iv[0]++) {                      ( 3)
  for( iv[1]=0; iv[1]<=5; iv[1]++) {                    ( 4)
    a[iv] = OP1;                                        ( 5)
  }                                                     ( 6)
}                                                       ( 7)
for( iv[0]=6; iv[0]<=9; iv[0]++) {                      ( 8)
  for( iv[1]=0; iv[1]<=5; iv[1]++) {                    ( 9)
    a[iv] = OP2;                                        (10)
  }                                                     (11)
}                                                       (12)
for( iv[0]=0; iv[0]<=9; iv[0]++) {                      (13)
  for( iv[1]=6; iv[1]<=9; iv[1]++) {                    (14)
    a[iv] = OP3;                                        (15)
  }                                                     (16)
}                                                       (17)
```

⇓Transformation into Canonical Order

```
a = MALLOC( [10,10]);                                   ( 1)
iv = MALLOC( [2]);                                      ( 2)
for( iv[0]=0; iv[0]<=5; iv[0]++) {                      ( 3)
  for( iv[1]=0; iv[1]<=5; iv[1]++) {                    ( 4)
    a[iv] = OP1;                                        ( 5)
  }                                                     ( 6)
  for( iv[1]=6; iv[1]<=9; iv[1]++) {                    ( 7)
    a[iv] = OP3;                                        ( 8)
  }                                                     ( 9)
}                                                       (10)
for( iv[0]=6; iv[0]<=9; iv[0]++) {                      (11)
  for( iv[1]=0; iv[1]<=5; iv[1]++) {                    (12)
    a[iv] = OP2;                                        (13)
  }                                                     (14)
  for( iv[1]=6; iv[1]<=9; iv[1]++) {                    (15)
    a[iv] = OP3;                                        (16)
  }                                                     (17)
}                                                       (18)
```

Fig. 16. A simple example for dense generators.

To fix this problem, the third generator has to be split up in the middle and to be fused with the other generators. The result of this transformation is shown at the bottom of figure 16. The third nesting has vanished. Instead, the other two generators contain copies of its body (lines (7)–(9) and (15)–(17)).

To formalize this transformation process, the two loop modifications needed are presented in figure 17. The operation $\mathscr{S}\mathtt{plit}$ effects *Loop Splitting* (Wolfe, 1995). It splits a given loop over an index range [`l`,`u`] up into two loops with identical bodies over two adjacent index ranges [`l`,`m`—1] and [`m`,`u`], provided that $\mathtt{l} \leqslant \mathtt{m} \leqslant \mathtt{u}$. The second operation, called $\mathscr{M}\mathtt{erge}$, applies *Loop Fusion* (Wolfe, 1995) to the

```
$\mathscr{S}plit$ $\llbracket$ for( iv[i]=l; iv[i]<=u; iv[i]++){
              Body;
            }                                      $\rrbracket$m

        = { for( iv[i]=l; iv[i]<=m-1; iv[i]++ ) {
              Body;
            }
            for( iv[i]=m; iv[i]<=u; iv[i]++ ) {
              Body;
            }

$\mathscr{M}erge$ $\llbracket$ for( iv[i]=l; iv[i]<=u; iv[i]++){ ,  for( iv[i]=l; iv[i]<=u; iv[i]++){ $\rrbracket$
              $Body_1$;                              $Body_2$;
            }                                    }

        = { for( iv[i]=l; iv[i]<=u; iv[i]++ ) {
              $Body_1$;
              $Body_2$;
            }
```

Fig. 17. The loop transformations $\mathscr{S}plit$ and $\mathscr{M}erge$.

```
Canon LoopNests
  = case LoopNests of {
      []        | true    -> []
      [ln:lns] | $\exists$ ln' $\in$ lns : (range ln) == (range ln')
                          -> Canon ( $\mathscr{M}erge\llbracket$ ln , ln' $\rrbracket$ ++ (lns \\ ln'))
      [ln:lns] | $\exists$ ln' $\in$ lns : (lower ln) < (lower ln') $\leqslant$ (upper ln)
                          -> Canon ( $\mathscr{S}plit\llbracket$ ln $\rrbracket$(lower ln') ++ lns)
      [ln:lns] | $\exists$ ln' $\in$ lns : (lower ln) $\leqslant$ (upper ln') < (upper ln)
                          -> Canon ( $\mathscr{S}plit\llbracket$ ln $\rrbracket$(upper ln') ++ lns)
      [ln:lns] | $\exists$ ln' $\in$ lns : (lower ln') < (lower ln) $\leqslant$ (upper ln')
                          -> Canon ( ([ln:lns] \\ ln') ++ $\mathscr{S}plit\llbracket$ ln' $\rrbracket$(lower ln))
      [ln:lns] | $\exists$ ln' $\in$ lns : (lower ln') $\leqslant$ (upper ln) < (upper ln')
                          -> Canon ( ([ln:lns] \\ ln') ++ $\mathscr{S}plit\llbracket$ ln' $\rrbracket$(upper ln))
      [ln:lns] | true     -> sort ( [ ForHeader { sort ( Canon Body) } ]
                                    ++ (Canon lns) )
                             where ForHeader { Body } = ln
  }
```

Fig. 18. Algorithm for transforming naïvely compiled code into canonical order.

outer loop[10]. It combines two loops with identical index ranges into a single one that contains both loop bodies. It should be noted here that this loop transformation in fact changes the order in which the result array is computed. Whereas this can safely be done for WITH-loops, in a more general setting an analysis would be required to make sure that any existing dependencies between the two loop bodies involved can still be observed after the fusion (Wolfe, 1995).

With these two operations at hand, an algorithm can be defined that systematically transforms naïvely compiled code into canonical order. Figure 18 gives such an algorithm in pseudo functional notation. It uses list notation to represent sequences of loop nestings as well as the usual list operations such as (++) for catenation

[10] It is called $\mathscr{M}erge$ here, since for non-dense generators a more general functionality than Loop Fusion is required.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

```
                                              for( iv[i] = l_i; iv[i] <= u_i; ) {
                                                stop = MIN( iv[i]+c_1-1, u_i);
                                                for( ; iv[i] <= stop; iv[i]++) {
for( iv[i] = l_i; iv[i] <= u_i; ) {               Body_1
  c_1 × Body_1                                  }
  ⋮                                     =_s     ⋮
  c_n × Body_n                                  stop = MIN( iv[i]+c_n-1, u_i);
}                                               for( ; iv[i] <= stop; iv[i]++) {
                                                  Body_n
                                                }
                                              }
```

Fig. 19. General form of loop nestings.

and (\\) for difference. The algorithm operates as follows: For each loop nesting *ln* found, the outermost index range is inspected:

- If within the remaining loop nestings (*lns*) a loop nesting *ln*′ with an identical range in the outermost loop is found (denoted by `(range` *ln*`) == (range` *ln*′`)`), these two nestings are merged wrt. the outer dimension and the transformation process continues on the new list of loop nestings.
- If within the remaining loop nestings (*lns*) a loop nesting *ln*′ is found whose range in the outermost loop overlaps that of the actual loop nesting *ln*, i.e. either the lower bound of *ln*′ (denoted by `lower` *ln*′) or the upper bound of *ln*′ (denoted by `upper` *ln*′) is within the range of *ln*, or vice versa, the overlapping generator is split up accordingly.
- Finally, if the loop nesting *ln* within the outermost dimension is disjoint from all other loop nestings, the transformation process is applied to the body of *ln* which, due to potential applications of $\mathscr{M}$`erge`, may consist of several loop nestings by itself. To ensure canonical order, the remaining loop nestings on each level have to be sorted wrt. the index ranges they are applied to, which is indicated by applications of `sort` within the recursive calls.

Although the algorithm shown in figure 18 was derived for dense generators only, it can be applied to non-dense generators as well. All there needs to be done is to find a more general form of loop pattern which – when applied recursively – (i) comprises loop nestings as they are generated from non-dense generators, and (ii) is general enough to be closed under applications of split and merge operations. Such a loop pattern is shown in figure 19. It consists of an outer loop, which determines the overall range in the actual dimension (i), and $n \in \mathbb{N}$ inner loops which contain n potentially different loop bodies. This allows not only to denote simple non-dense generators, it also allows for loop nestings that contain arbitrary periodical sequences of sub-loops. The overall period is determined by the sum of the individual ranges of all inner loops ($c_1 ... c_n$). In order to prevent an inner loop from exceeding the overall upper boundary u_i, the upper limit of all inner loops is computed as a minimum of the intended range and u_i. To improve program readability inner loops are abbreviated by expressions of the form *number* × *body*,

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

$$\mathscr{S}\mathtt{plit}\left[\!\!\left[\begin{array}{l}\texttt{for(iv[0] = 0; iv[0] <= 1023;) \{}\\ \quad \texttt{3} \times \mathit{Body}_1\\ \quad \texttt{7} \times \mathit{Body}_2\\ \texttt{\}}\end{array}\right]\!\!\right]\texttt{65} = \left\{\begin{array}{l}\texttt{for(iv[0] = 0; iv[0] <= 64;) \{}\\ \quad \texttt{3} \times \mathit{Body}_1\\ \quad \texttt{7} \times \mathit{Body}_2\\ \texttt{\}}\\ \texttt{for(iv[0] = 65; iv[0] <= 1023;) \{}\\ \quad \texttt{5} \times \mathit{Body}_2\\ \quad \texttt{3} \times \mathit{Body}_1\\ \quad \texttt{2} \times \mathit{Body}_2\\ \texttt{\}}\end{array}\right.$$

Fig. 20. An example for splitting a loop nesting in its general form.

$$\mathscr{M}\mathtt{erge}\left[\!\!\left[\begin{array}{l}\texttt{for(iv[0] = 0; iv[0] <= 1023;) \{}\\ \quad \texttt{3} \times \mathit{Body}_1\\ \quad \texttt{7} \times \mathit{Body}_2\\ \texttt{\}}\end{array}, \begin{array}{l}\texttt{for(iv[0] = 0; iv[0] <= 1023;) \{}\\ \quad \texttt{3} \times \mathit{Body}_3\\ \quad \texttt{2} \times \mathit{Body}_4\\ \texttt{\}}\end{array}\right]\!\!\right] = \left\{\begin{array}{l}\texttt{for(iv[0] = 0; iv[0] <= 1023;) \{}\\ \quad \texttt{3} \times \mathit{Body}_1\texttt{;}\ \mathit{Body}_3\\ \quad \texttt{2} \times \mathit{Body}_2\texttt{;}\ \mathit{Body}_4\\ \quad \texttt{3} \times \mathit{Body}_2\texttt{;}\ \mathit{Body}_3\\ \quad \texttt{2} \times \mathit{Body}_2\texttt{;}\ \mathit{Body}_4\\ \texttt{\}}\end{array}\right.$$

Fig. 21. An example for merging two loop nestings in their general form.

indicating a loop that ranges over *number* elements and has a body *body*. The left hand side of figure 19 shows the loop pattern in abbreviated form.

All there remains to be done is to redefine $\mathscr{S}\mathtt{plit}$ and $\mathscr{M}\mathtt{erge}$ on the more general loop pattern. However, this task turns out to be more complex than in the dense case. These operations are therefore explained by means of examples here. Formal definitions can be found in Appendices A and B, respectively.

Figure 20 shows an example application of $\mathscr{S}\mathtt{plit}$. The outer loop of the loop nesting to be split at position `65` ranges from `0` to `1023`. As for the dense case, the loop nesting is split into two almost identical loop nestings, whose outer loops range from `0` to `64`, and from `65` to `1023`. However, due to the existence of inner loops, the relation between the period of these loops ($\texttt{3} + \texttt{7} = \texttt{10}$ in the example) and the actual splitting index (`65`) affects the form of the second loop. Since `65` is not an integral multiple of `10`, the inner loops have to be rotated by `5` elements which requires the second inner loop to be split up as well. As a result, the ‘last’ `5` instances of Body_2 constitute the first inner loop, followed by the `3` instances of Body_1 and the ‘first’ `2` instances of Body_2.

An example for an application of $\mathscr{M}\mathtt{erge}$ to two loop nestings in general form is given in figure 21. In contrast to the dense case, the bodies of the outer loops can not simply be appended to each other since the inner loops of both of them contain increments of the index vector component `iv[0]`. They have to be synchronized which requires adapting the ranges of the inner loops and thus adapting both periods

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

```
Fitting LoopNests
  = case LoopNests of {
      []      | true    -> []
      [ln:lns]| ((upper ln) - (lower ln)) mod (period ln) = m ≠ 0
                        -> Fitting ( 𝒮plit⟦ ln ⟧((upper ln) - m) ++ lns)
      [ln:lns]| true    -> [ ForHeader { Fitting Body } ] ++ (Fitting lns)
                           where ForHeader { Body } = ln
    }
```

Fig. 22. Fitting algorithm.

as well. The resulting loop nesting has a period of 10 which is the least common multiple of the individual periods (5 and 10). It is split up into four inner loops of sizes 3 and 2 which are primarily derived from the two inner loops of the second loop nesting. The bodies of the inner loops actually contain the concatenated loop bodies taken from the two different loop nestings to be merged.

As can be seen from the examples discussed so far, even simple non-dense generators may lead to rather complex loop nestings when transformed into canonical order. In particular, the number of inner loops inserted in each dimension usually grows when transforming loop nestings into canonical order. The upper bound of these inner loops is computed as a minimum of the intended range and the upper bound of the surrounding outer loop. These minimum computations do not only introduce overhead in all but the last iterations of the outer loop, they also prevent the C compiler from unrolling inner loops. To avoid these deficiencies, a simple optimization technique called *Fitting* is applied. The basic idea is to split each loop after the last integral multiple of the period within the range to be traversed. A description of this optimization in pseudo functional notation is shown in figure 22. Similar to the algorithm Canon for each loop nesting *ln* the outermost range is inspected. Whenever the range of the outer loop ((upper *ln*) - (lower *ln*)) is not an integral multiple of the period (referred to by (period *ln*)), the last incomplete period is split off and fitting proceeds. Otherwise, fitting is propagated into all bodies of the inner loops and into the remaining loop nestings as well.

After this optimization has been applied, all minimum computations can be eliminated which in turn allows the range of all inner loops to be determined statically.

The canonical order established so far only ensures spatial reuse for the write accesses. To further improve the cache behavior, other loop transformations have to be applied that take into account the cache characteristics of the intended target hardware. See Grelck (2001) for details.

5 Case study: the PDE1-benchmark

In this section, the suitability of SaC for implementing numerically intensive applications is investigated. Rather than trying to present a systematic performance evaluation, we intend to give a flavor of SaC from the programmer's point ov view. A benchmark algorithm is used as a vehicle to study the various aspects of SaC as an implementation language, e.g. expressiveness of the language constructs, potential for modular specifications, and runtime efficiency.

The program under consideration is the so-called PDE1 benchmark which originates from various HPF compiler comparisons. It implements a so-called red-black relaxation algorithm for approximating three-dimensional poisson equations and thus can be seen as a typical application kernel for many number crunching applications.

In the first subsection, the PDE1 algorithm is introduced along with the essential parts of its HPF implementation. Starting out from a straightforward re-implementation of the HPF solution in SAC, section 5.2 discusses aspects of software reuse and program readability in the context of shape-invariant programming facilities. In the course of this discussion, six different variants of the SAC implementation with an increasing level of abstraction are proposed, ending up with an entirely shape-invariant APL-style solution. The runtimes of all these implementations are examined in Subsection 5.3. They are contrasted with the runtimes obtained from the HPF implementation using the SUN Fortran95 compiler.

5.1 PDE1 – the given algorithm

The core of the PDE1 algorithm is a so-called stencil operation on a three-dimensional array. It iteratively re-computes the inner elements of an array, i.e. all elements with non-minimal/non-maximal indices, as weighted sum of adjacent elements. In the case of PDE1, the stencil is almost trivial: the value of an element $u'_{i,j,k}$ is computed by adding up all six direct neighbors of $u_{i,j,k}$, adding that value to a fixed number $h^2 f_{i,j,k}$, and subsequently multiplying with a constant factor. Assuming NX, NY, and NZ to denote the extents of the three-dimensional arrays U, U1, and F, this operation can in HPF be specified as:

```
    U1(2:NX-1,2:NY-1,2:NZ-1) =                               &
  &               FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+       &
  &    U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+        &
  &    U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+        &
  &    U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))
```

The central language feature of Fortran90/HPF used here is the so-called triple notation. It allows triples of the form $l : u : s$ or tuples of the form $l : u$ to be used insted of single indices, which affects assignments to refer to entire index ranges (all elements between l and u strided by s) rather than to single elements. Thus, the assignment above in fact denotes assignments to all inner elements of U1.

However, in the case of PDE1, this operation is not applied to all elements in a single step, but in two consecutive steps on two disjoint sets of elements. These sets are called the *red* set and the *black* set, which include all those elements with even and odd indices in the first dimension, respectively. In HPF, this can be expressed by introducing a three-dimensional array of booleans RED whose elements are .TRUE. for all those elements belonging to the red set and .FALSE. for all the black elements.

```
RED(2:NX-1:2,2:NY-1,2:NZ-1) = .TRUE.
RED(3:NX-1:2,2:NY-1,2:NZ-1) = .FALSE.
```

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

With this definition at hand, relaxation on the red set can be specified using the `WHERE` construct of HPF:

```
        WHERE(RED(2:NX-1,2:NY-1,2:NZ-1))
          U1(2:NX-1,2:NY-1,2:NZ-1) =                          &
     &               FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+     &
     &    U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+      &
     &    U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+      &
     &    U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))

        END WHERE

        WHERE(RED(2:NX-1,2:NY-1,2:NZ-1))
           U (2:NX-1,2:NY-1,2:NZ-1) = U1 (2:NX-1,2:NY-1,2:NZ-1)
        END WHERE
```

Note, that the first WHERE-block initializes only elements of `U1` that belong to the red set. To make sure that all non-red elements of the result remain the same, the freshly computed elements of `U1` are copied back into `U` by means of a second WHERE-block. The complete relaxation algorithm of PDE1 consists of an iteration loop that contains two of the blocks above, one for the red elements and another one for the black elements.

5.2 Implementing PDE1 in SAC

The HPF solution can be carried over to SAC almost straightforwardly. Rather than using the triple notation of HPF, the computation of the inner elements is in SAC specified for a single element at index position `iv`, which by a WITH-loop is mapped on all inner elements:

```
red = with ( [1,0,0] <= iv < shape(u) step [2,1,1])
      genarray ( shape(u), iv, true);

u = with (. < iv < . ) {
      if( red[iv]) {
        local_sum = u[iv+[1,0,0]] + u[iv-[1,0,0]]
                  + u[iv+[0,1,0]] + u[iv-[0,1,0]]
                  + u[iv+[0,0,1]] + u[iv-[0,0,1]];              (Low-Level)
        val = factor * (hsq * f[iv] + local_sum);
      } else {
        val = u[iv];
      }
    } modarray (u, iv, val);
```

Instead of the WHERE-construct in HPF, an explicit conditional is inserted into the body of the WITH-loop which computes the weighted sum of adjacent elements. This solution does not only specify the computation of the red elements, but the alternative part of the conditional includes a specification for the black elements as well. As a consequence, there is no need to 'copy' the red elements as it is required in the HPF solution. Instead, the resulting array may directly be named `u` again.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

However, this solution has several weaknesses. First of all, the relaxation part and the distinction between red and black elements are tightly coupled, which impairs program readability and software reuse. To set them apart, the selective application of the relaxation step has to be abstracted out into an array operation of its own. This can be done by defining a `Where`-operation in SAC:

```
double[*] Where( bool[*] mask, double[*] a, double[*] b)
{
  c = with( .<= iv <= . ) {
          if( mask[iv]) val = a[iv];
          else val = b[iv];
        } genarray( shape(a), val);
  return( c);
}
```

It takes three arrays as arguments: an array `mask` of booleans, and two arrays `a` and `b` of doubles. Provided that all three arrays are of identical shape, a new array `c` of the same shape is created, whose elements are copied from those of the array `a` if the mask evaluates to `true`, and from `b` otherwise.

Assuming a function `Relax` to implement relaxation on all inner elements of an array, red-black relaxation now can be defined in terms of that operation:

```
red = with ( [1,0,0] <= iv <= . step [2,1,1])
      genarray ( shape(u), iv, true);
                                                                    (PDE1)
u = Where( red, Relax(u, f, hsq), u);
u = Where( !red, Relax(u, f, hsq), u);
```

Note here that the black set is referred to by `!red`, i.e. by using the element-wise extension of the negation operator (`!`).

This specification has several advantages over the low-level specification: it is much more concise, it does not require the weighted sum to be specified twice, and the intended functionality is more clearly exposed. Furthermore, the specification of `Relax` may be reused in other relaxation-based contexts.

An implementation of the relaxation step can be derived straightforwardly by abstracting out the computation of the weighted sum of neighbor elements:

```
double[*] Relax( double[*] u, double[*] f, double hsq)
{
  factor = 1d/6d;
  u1 = with (. < iv < . ) {
         local_sum = u[iv+[1,0,0]] + u[iv-[1,0,0]]
                   + u[iv+[0,1,0]] + u[iv-[0,1,0]]              (Relax 1)
                   + u[iv+[0,0,1]] + u[iv-[0,0,1]];
         val = factor * (hsq * f[iv] + local_sum);
       } modarray (u, iv, val);
  return( u1);
}
```

Note that the usage of `<` instead of `<=` on both sides of the generator part restricts the elements to be computed to the inner elements of the array `u`.

The disadvantage of this solution, which in the sequel will be referred to as (Relax 1) is that it is tailor-made for the given stencil. In the same way the access triples in the HPF-solution have to be adjusted whenever the stencil changes, the offset vectors have to be adjusted in the SAC solution. These adjustments are very error-prone, particularly, if the size of the stencil increases or the dimensionality of the problem changes. These problems may be remedied by abstracting from the problem specific part with the help of an array of weights W. In this particular example, W is an array of shape [3,3,3] whose elements are all 0.0 except the six direct neighbor elements of the center element, which are set to 1.0. Relaxation thus can be defined as:

```
double[*] Relax( double[*] u, double[*] f, double hsq, double[*] W)
{
  factor = 1d/6d;
  u1 = with (. < iv < . ) {
         block = tile( shape(W), iv-1, u);
         local_sum = sum( W * block);
       } modarray( u, iv, factor * (hsq * f[iv] + local_sum));
  return( u1);
}
```

(Relax 2)

For each inner element of u1 in this piece of program a sub-array block is taken from u which holds all the neighbor elements of u[iv]. This is done by applying the library function tile(*shape*, *offset*, *array*), which creates an array of shape *shape* whose elements are taken from *array* starting at position *offset*. The weighted sum of neighbor elements may be computed by an application sum(W * block), where (W * block) multiplies element by element the arrays W and block and sum sums up all elements of this product array.

Abstracting from the problem specific stencil data has another advantage: the resulting program does not only support arbitrary stencils but can also be applied to arrays and stencils of other dimensionalities without changes, for which the usage of shape(W) rather than [3,3,3] as first argument for tile is essential.

Though the indexing operations have been eliminated by introducing W, the specification still consists of a problem specific WITH-loop which contains an element-wise specification of the relaxation step. This includes some redundancy, since parts of the functionality, e.g. the multiplication of the constant factor with each element of the result, already exist as library functions. Extensive usage of these library functions allows these operations to be 'lifted' out of the body of the with-loop:

```
double[*] Relax( double[*] u, double[*] f, double hsq, double[*] W)
{
  factor = 1d/6d;
  u1 = hsq * f;
  u1 += with (. < iv < . ) {
          block = tile( shape(W), iv-1, u);
        } modarray( u1, iv, sum( W * block));
  u1 = CombineInnerOuter( factor * u1, u);
  return( u1);
}
```

(Relax 3)

Since these functions effect all elements rather than just the inner ones, a new function CombineInnerOuter(*inner*, *outer*) is required to adjust the border elements. It takes two identically shaped arrays inner and outer as arguments to create a new one whose inner elements are taken from *inner* and whose outer elements are taken from *outer*. A shape-invariant version of this function can be specified by a single WITH-loop:

```
double[*] CombineInnerOuter(double[*] inner, double[*] outer)
{
  res = with ( . < iv < .)
        modarray( outer, iv, inner[iv]);
  return( res);
}
```

Taking further the idea of lifting operations out of the body of the WITH-loop and applying more generally applicable array operations instead, leads to yet another approach for specifying the stencil operation. Instead of addressing neighbor elements and summing them up on the level of scalars, the entire array could be shifted and subsequently summed up. This yields the following specification

```
double[*] Relax( double[*] u, double[*] f, double hsq)
{
  factor = 1d/6d;
  u1 = hsq * f;

  for (i=0; i<dim(u); i++) {
    u1 += shift( i, 1, 0d, u);                              (Relax 4)
    u1 += shift( i, -1, 0d, u);
  }
  u1 = CombineInnerOuter( factor*u1, u);

  return( u1);
}
```

where shift(*dim*, *num*, *value*, *array*) shifts the elements of *array* with respect to the axis *dim* by *num* elements towards increasing indices.

Again, this specification can be made invariant against different stencils by introducing an array W of weights and summing up shifted arrays by means of a fold WITH-loop.

```
double[*] Relax( double[*] u, double[*] f, double hsq, double[*] W)
{
  factor = 1d/6d;

  u1 = with ( 0*shape(W) <= iv < shape(W))
       fold( +, hsq * f, W[iv] * shift( 1-iv, 0d, u));      (Relax 5)

  u1 = CombineInnerOuter( u1*factor, u);

  return( u1);
}
```

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

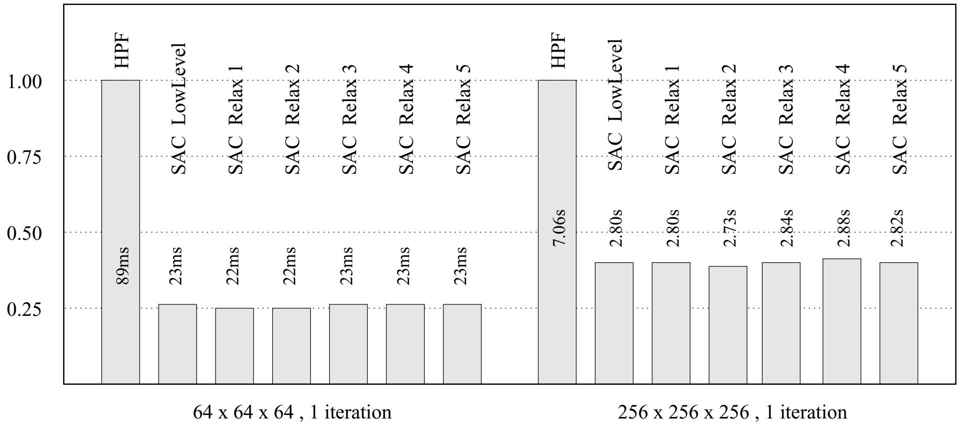


Fig. 23. Runtimes for red black relaxation implementations in HPF and in SAC.

Here another slightly different `shift` operation from the SAC-library is used. Rather than shifting an array with respect to one axis, `shift(` *`shift_vec`*`,` *`val`*`,` *`array`*`)` shifts *array* with respect to all axes. The number of positions to be shifted per axis is defined through the vector *shift_vec*. Another peculiarity of this specification is the usage of `hsq * f` as 'neutral element' for the fold operation. This integrates elegantly `hsq * f` as initial summand of the element-wise addition of arrays.

5.3 A runtime comparison with HPF

In this subsection, the runtimes for the six SAC implementations of the PDE1 benchmark discussed in the previous subsection are contrasted with the runtime of the HPF implementation whose essential parts have been given in section 5.1.

All measurements are made on a SUN Ultra2 Enterprise 450 with 4 GB of memory, running SOLARIS-8. The C code generated by the SAC compiler is compiled to native code by the SUN Workshop 6 compiler cc v5.1. For the HPF implementation, the SUN Workshop 6 compiler f90 v 6.1 is used.

To allow for a fair comparison, the HPF implementation is taken from the demo benchmarks that come with the ADAPTOR HPF compiler release (Brandes & Zimmermann, 1994). It contains timer calls that measure the time spent in the numerical part only. Since such timer calls are not available in SAC, all runtimes are derived from measuring wall clock times for two different numbers of iterations and dividing the runtime difference by the difference of the numbers of iterations.

Figure 23 shows the runtimes for two different problem sizes of the arrays to be iterated: 64^3 elements (≈ 2MB per array) and 256^3 elements (≈ 130MB per array). All runtimes are shown relative to the HPF runtime. The average times for a single iteration on the entire array (red and black set) are written into the bars.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

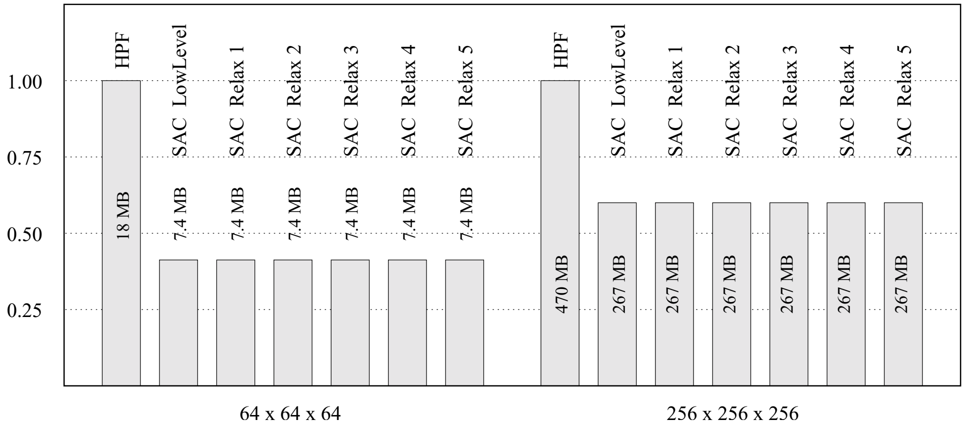


Fig. 24. Memory consumptions for red black relaxation implementations in Hpf and in SaC.

It can be observed that all SaC variants outperform the Hpf code by factors of roughly 4 in the 64^3 elements case, and factors of roughly 2.5 in the 256^3 elements case. Despite the considerable difference in their level of abstraction, all six SaC variants perform almost identical. When examining the C code generated by the SaC compiler it turns out that all versions – apart from variable renaming – are almost identical. Only minor variations can be observed in the way the neighbor elements of an array are accessed. The C code also reveals that the array `f` of constants as well as the mask array `red` are entirely eliminated by the SaC compiler which, most likely, is the main reason for the performance edge over the Hpf solution. In fact, this may also explain the difference between the speedup factors for the two example sizes, as for smaller overall memory demands the potential gains due to better cache reuse substantially increase.

This is reflected by the memory consumption shown in figure 24 which was measured by using the system command `top`. Again, the memory demands are shown relative to the memory demand of the Hpf version, and the absolute numbers are annotated in the bars. In particular, for the 256^3 problem size where a single array requires $\approx$130MB of memory, the memory demand can be readily related with the program sources. The Hpf implementation requires roughly 3.5 times the space of a single array of doubles: two arrays serving as source and destination of every relaxation step, one array for holding the values of `f`, and an arrays of booleans for the mask which requires half the size of arrays of doubles. In contrast, the SaC versions require only 2 times the space of a single array of doubles due to the elimination of `f` and `red`.

The elimination of `f` and `red` as well as the transformation of all SaC versions into almost identical C programs can be primarily attributed to *With Loop Folding*. To make its effects visible, figure 25 shows runtimes obtained when explicitly turning off that particular optimization. Irrespective of the problem size, the runtimes of the various SaC versions increase with the level of abstraction applied. An examination

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

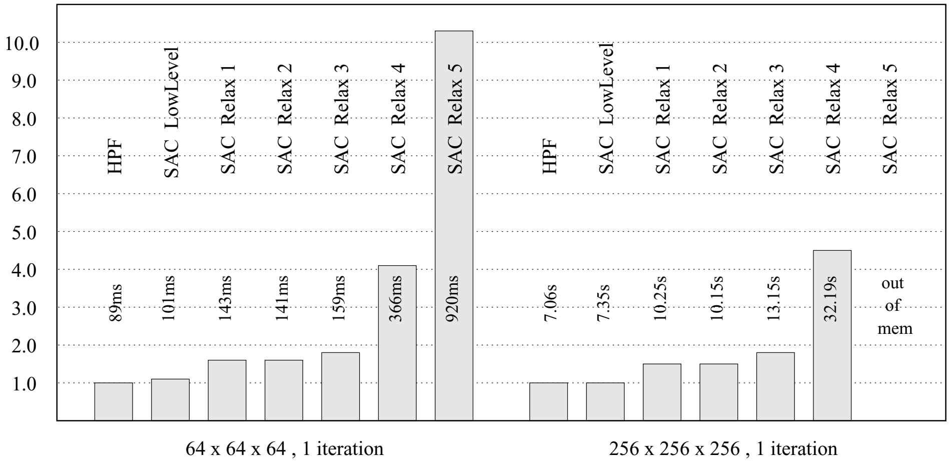


Fig. 25. Runtimes with disabled *With Loop Folding*.

of the C code confirms that they only differ wrt. temporary arrays being created. Without *With Loop Folding*, the low-level version creates arrays `f` and `red`, which reduces the factor over HPF from 4 to roughly 0.9. The versions (Relax 1) and (Relax 2) now compute a relaxation step on the entire array before restricting the result to the red (black) set by means of an application of `Where`. However, the runtimes of these versions are still only about a factor of 1.5 slower than those of the HPF implementations. (Relax 3) requires further temporaries for the simple arithmetic operations that are lifted out of the inner loop, which leads to a less favorable factor. A more significant slowdown can be observed for the versions (Relax 4) and (Relax 5). Since they replace the selection of neighbor elements by shifts of the entire array, it is copied several times while doing a single relaxation step. This leads to runtimes which are roughly 4 and 10 times slower than the HPF solution.

6 Conclusion

This paper presents a novel concept for supporting a shape-invariant programming style for array operations in functional languages. The primary objectives of this approach are (i) to provide a level of abstraction which liberates array programming from the details of specifying starts, stops and strides of iteration loops and of artful loop nestings, and (ii) to demonstrate that these high level specifications can be compiled to executable code whose runtime efficiency is competitive with that obtained by compilation of equivalent HPF programs, thus closing the performance gap between functional and imperative programming in the number crunching department.

This concept, which has been implemented as an integral part of a fairly simple functional language called SaC (for Single Assignment C) with a call-by-value semantics, is based on the representation of arrays by shape and data vectors

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

and on compound array operations specified in terms of shape-invariant array comprehension constructs called WITH-loops. The strength of this approach largely derives from the fact that the WITH-loops are perfect vehicles for implementing, besides a set of standard operations as they are supported by state-of-the-art array programming languages such as APL, customized sets of application-specific array operations, both of which may be linked as libraries to SAC executables. With a well chosen set of customized operations at hand, application programs may be written in fairly concise and comprehensible form which, following the basic idea of the functional paradigm, ensures correct programs largely by construction based on abstractions.

The high-level programming style of SAC is complemented by a sophisticated type system and by refined compilation techniques to generate highly efficient executable code.

The type system is based on a hierarchy of array types with increasingly specific shape information which allows an inference algorithm to step-wise turn shape-invariant into shape-specific programs. Though shape inference generally is an undecidable problem, decidability of the type system of SAC is ensured by falling back on more general shapes and inserting into the code dynamic type checks, if it otherwise would fail. As a cosequence, inferred array types are unique modulo subtyping only.

Exact knowledge about shapes being an essential pre-requisite for efficient array computations, code optimizations primarily focusses on a new technique called *With Loop Folding* which rigorously eliminates the generation of intermediate arrays from compositions of compound operations. Since all array operations are implemented as WITH-loops, this optimization plays the key role in generating fast code.

A particularly difficult problem of the compilation into C code which is extensively discussed in the paper concerns efficient ways and means of handling the application of different operations to disjoint sections of an array. Since naïve compilation would generate memory (and cache) access patterns that match less than perfectly with the underlying machinery, a great deal of code optimizations must be devoted to rearranging array traversals so as to minimize cache misses. Here it pays off that SAC is a functional language which allows to perform operations on individual array elements in any order since there are no side effects to worry about.

The elegance of array programming that comes with the array concept of SAC and the runtime efficiency of SAC executables are demonstrated by means of a red-black relaxation program taken from the APR HPF-benchmarks. The HPF implementation is compared with several variants of SAC implementations that feature increasing levels of abstraction. The runtime figures show that all SAC implementations, different degrees of abstraction notwithstanding, execute in about the same time, very likely due to the overriding effect of *With Loop Folding*, but outperform the HPF implementation by factors better than 2.5, depending on problem sizes.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

A Definition of $\mathscr{S}plit$ for loop nestings in their general form

$$\mathscr{S}\mathtt{plit}\left[\!\!\left[\begin{array}{l}\mathtt{for(\ iv[}i\mathtt{]\ =\ }l\mathtt{;\ iv[}i\mathtt{]\ <=\ }u\mathtt{;\ )\ \{}\\ \quad c_1 \ \times \ \mathit{Body}_1\\ \quad \vdots\\ \quad c_n \ \times \ \mathit{Body}_n\\ \mathtt{\}}\end{array}\right]\!\!\right]m$$

$$= \left\{\begin{array}{l}\mathtt{for(\ iv[}i\mathtt{]\ =\ }l\mathtt{;\ iv[}i\mathtt{]\ <=\ }m-1\mathtt{;\ )\ \{}\\ \quad c_1 \ \times \ \mathit{Body}_1\\ \quad \vdots\\ \quad c_n \ \times \ \mathit{Body}_n\\ \mathtt{\}}\\ \mathtt{for(\ iv[}i\mathtt{]\ =\ }m\mathtt{;\ iv[}i\mathtt{]\ <=\ }u\mathtt{;\ )\ \{}\\ \quad c_{k_2} \ \times \ \mathit{Body}_k\\ \quad c_{k+1} \ \times \ \mathit{Body}_{k+1}\\ \quad \vdots\\ \quad c_n \ \times \ \mathit{Body}_n\\ \quad c_1 \ \times \ \mathit{Body}_1\\ \quad \vdots\\ \quad c_{k-1} \ \times \ \mathit{Body}_{k-1}\\ \quad c_{k_1} \ \times \ \mathit{Body}_k\\ \mathtt{\}}\end{array}\right.$$

where

$$\begin{array}{l}\sum_{i=1}^{k-1} c_i \leqslant m' < \sum_{i=1}^{k} c_i\\ m' = (m-l)\ \mathtt{mod}\ \sum_{i=1}^{n} c_i\\ c_{k_1} = m' - \sum_{i=1}^{k-1} c_i\\ c_{k_2} = c_k - c_{k_1}\end{array}$$

B Definition of $\mathscr{M}erge$ for loop nestings in their general form

$$\mathscr{M}erge\left[\!\!\left[\begin{array}{l}\mathtt{for(\ iv[}i\mathtt{]\ =\ }l\mathtt{;\ iv[}i\mathtt{]\ <=\ }u\mathtt{;\ )\ \{}\\ \quad \mathit{Body}_1\\ \mathtt{\}}\end{array}\ ,\ \begin{array}{l}\mathtt{for(\ iv[}i\mathtt{]\ =\ }l\mathtt{;\ iv[}i\mathtt{]\ <=\ }u\mathtt{;\ )\ \{}\\ \quad \mathit{Body}_2\\ \mathtt{\}}\end{array}\right]\!\!\right]$$

$$= \left\{\begin{array}{l}\mathtt{for(\ iv[}i\mathtt{]\ =\ }l\mathtt{;\ iv[}i\mathtt{]\ <=\ }u\mathtt{;\ )\ \{}\\ \quad \mathscr{M}erge'\left[\!\!\left[\left.\begin{array}{c}\mathit{Body}_1\\ \vdots\\ \mathit{Body}_1\end{array}\right\}m_1\ ,\ \left.\begin{array}{c}\mathit{Body}_2\\ \vdots\\ \mathit{Body}_2\end{array}\right\}m_2\right]\!\!\right]\\ \mathtt{\}}\end{array}\right.$$

where

$$m_1 = \frac{lcm(p_1,p_2)}{p_1}\ ,\ \ m_2 = \frac{lcm(p_1,p_2)}{p_2}\ ,\ \ p_i = \mathtt{period}(\mathit{Body}_i)$$

$$\mathcal{M}erge'\left[\!\!\left[\begin{array}{l} c_1 \times \mathtt{Body}_1 \\ \quad \mathtt{Rest}_1 \end{array} , \begin{array}{l} c_2 \times \mathtt{Body}_2 \\ \quad \mathtt{Rest}_2 \end{array} \right]\!\!\right]$$

$$= \left\{ \begin{array}{ll} \begin{array}{l} c_1 \times \left\{ \begin{array}{l} \mathtt{Body}_1 \\ \mathtt{Body}_2 \end{array} \right. \\ \mathcal{M}erge'[\mathtt{Rest}_1, \mathtt{Rest}_2] \end{array} & \mathtt{iff}\ c_1 = c_2 \\[2ex] \begin{array}{l} c_1 \times \left\{ \begin{array}{l} \mathtt{Body}_1 \\ \mathtt{Body}_2 \end{array} \right. \\ \mathcal{M}erge'\left[\!\!\left[\mathtt{Rest}_1 , \begin{array}{l} (c_2 - c_1) \times \mathtt{Body}_2 \\ \mathtt{Rest}_2 \end{array} \right]\!\!\right] \end{array} & \mathtt{iff}\ c_1 < c_2 \\[2ex] \begin{array}{l} c_2 \times \left\{ \begin{array}{l} \mathtt{Body}_1 \\ \mathtt{Body}_2 \end{array} \right. \\ \mathcal{M}erge'\left[\!\!\left[\begin{array}{l} (c_1 - c_2) \times \mathtt{Body}_2 \\ \mathtt{Rest}_1 \end{array} , \mathtt{Rest}_2 \right]\!\!\right] \end{array} & \mathtt{otherwise} \end{array} \right.$$

References

Achten, P. and Plasmeijer, R. (1993) *The Beauty and the Beast*. Technical report 93-03, University of Nijmegen.

Adams, J. C., Brainerd, W. S., Martin, J. T. *et al.* (1992) *Fortran90 Handbook – Complete ANSI/ISO Reference*. McGraw-Hill.

Allen, R. and Kennedy, K. (1987) Automatic translation of Fortran into vector form. *Toplas*, **9**(4), 491–542.

Allen, R. and Kennedy, K. (2001) *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann.

Appel, A. W. (1998) *Modern Compiler Implementation in C*. Cambridge University Press.

Augustsson, L. (1998) Cayenne – a language with dependent types. *Proceedings 3rd ICFP*. ACM Press.

Bacon, D. F., Graham, S. L. and Sharp, O. J. (1994) Compiler transformations for high-performance computing. *ACM Comput. Surv.* **26**(4), 345–420.

Baker, H. (1991) Shallow binding makes functional arrays fast. *ACM SIGPLAN Notices*, **26**(8), 145–147.

Barendregt, H. P. (1981) *The Lambda Calculus, its Syntax and Semantics*. Studies in logics and the foundations of mathmatics, **103**. North-Holland.

Bird, R. S. and Wadler, P. L. (1988) *Functional Programming*. Prentice Hall.

Brandes, T. and Zimmermann, F. (1994) ADAPTOR – a transformation tool for HPF programs. *Programming Environments for Massively Parallel Distributed Systems*, pp. 91–96. Birkhäuser Verlag.

Burke, C. (1996) *J and APL*. Iverson Software Inc., Toronto, Canada.

Cann, D. C. (1989) *Compilation Techniques for High Performance Applicative Computation*. Technical report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore, CA.

Cann, D. C. (1992) Retire Fortran? A debate rekindled. *Commun. ACM*, **35**(8), 81–89.

Castagna, G. (1995) Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst.* **17**(3), 431–447.

Chakravarty, M. M. T. and Keller, G. (2001) Functional array fusion. In: Leroy, X. (ed.), *Proceedings ICFP'01*. ACM Press.

Chin, W.-N. (1994) Safe fusion of functional expressions II: further improvements. *J. Functional Program.* **4**(4), 515–550.

Chin, W.-N. and Khoo, S.-C. (2000) Calculating sized types. *Proceedings ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pp. 62–72. Boston, MA. ACM Press.

Cohen, J. (1981) Garbage collection of linked data structures. *ACM Comput. Surv.* **13**(3), 341–367.

Coleman, S. and McKinley, K. (1995) Tile size selection using cache organization and data layout. *Proceedings ACM SIGPLAN'95 Conference on Programming Language Design and Implementation* (*PLDI*), pp. 279–290. La Jolla, CA.

Ding, C. (2000) *Improving Effective Bandwidth Through Compiler Enhancement of Global and Dynamic Cache Reuse.* PhD thesis, Rice University, Houston, TX.

Field, A. J. and Harrison, P. G. (1988) *Functional Programming*. Addison-Wesley.

Gao, G. R., Olsen, R., Sarkar, V. and Thekkath, R. (1993) Collective loop fusion for array contraction. In: Banerjee, U., Gelernter, P., Nicolau, A. and Padua, D. (eds.), *Proceedings 5th Workshop on Languages and Compilers for Parallel Computing: Lecture Notes in Computer Science 1993.* Springer-Verlag.

Gill, A. (1996) *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Glasgow University, UK.

Gopinath, K. and Hennessy, J. L. (1989) Copy elimination in functional languages. *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pp. 303–314. ACM Press.

Grelck, C. (2001) *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC – Single Assignment C.* PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel.

Grelck, C. and Scholz, S. B. (1995) Classes and objects as basis for I/O in SAC. In: Johnsson, T. (ed.), *Proceedings Workshop on the Implementation of Functional Languages'95*, pp. 30–44. Chalmers University.

Grelck, C., Kreye, D. and Scholz, S.-B. (2000) On code generation for multi-generator WITH-Loops in SAC. In: Koopman, P. and Clack, C. (eds.), *Proc. 11th International Workshop on Implementation of Functional Languages* (*IFL'99*)*: Lecture Notes in Computer Science 1868*, pp. 77–95. Lochem, The Netherlands. Springer-Verlag.

Hammes, J., Sur, S. and Böhm, W. (1997) On the effectiveness of functional language features: NAS benchmark FT. *J. Functional Program.* **7**(1), 103–123.

Hennessy, J. L. and Patterson, D. A. (1995) *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann.

High Performance Fortran Forum (1994) *High Performance Fortran Language Specification V1.1.*

Hindley, J. R. and Seldin, J. P. (1986) *Introduction to Combinators and Lambda Calculus.* London Mathematical Society Student Texts, vol. 1. Cambridge University Press.

Hudak, P. and Bloss, A. (1985) The aggregate update problem in functional programming systems. *POPL'85*, pp. 300–313. ACM Press.

Hughes, J., Pareto, L. and Sabry, A. (1996) Proving the correctness of reactive systems using sized types. *POPL'96*. ACM Press.

International Standards Organization (1984) *International Standard for Programming Language APL.* ISO n8485, ISO.

Iverson, K. E. (1962) *A Programming Language*. Wiley.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

Jay, C. B. and Steckler, P. A. (1998) The functional imperative: Shape! In: Hankin, C. (ed.), *Programming Languages and Systems: 7th European Symposium on Programming ESOP'98: Lecture Notes in Computer Science 1381*, pp. 139–53. Lisbon, Portugal. Springer-Verlag.

Jenkins, M. A. (1999) *Choices in Array Theory*. (In preparation.) Jenkins, M. A. and Falster, P. "Array Theory and Nial" Technical Report No. 157, 1999 Technical University of Denmark ELTEK, Lyngby, Denmark.

Jenkins, M. A. and Glagow, J. I. (1989) A logical basis for nested array data structures. *Comput. Languages J.* **14**(1), 35–51.

Jenkins, M. A. and Jenkins, W. H. (1993) *The Q'Nial Language and Reference Manuals*. Nial Systems Ltd., Ottawa, Canada.

Kluge, W. E. (1992) *The Organization of Reduction, Data Flow and Control Flow Systems*. MIT Press.

Kx Systems (1998) *K Reference Manual Version 2.0*. Kx Systems, Miami, Florida. <http://www.kx.com/technical/documents/kreflite.pdf>.

Lam, M. S., Rothberg, E. E. and Wolf, M. E. (1991) The cache performance of blocked algorithms. *Proceedings 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74. Palo Alto, CA.

Launchbury, J. and Peyton Jones, S. (1994) Lazy functional state threads. *Programming Languages Design and Implementation*. ACM Press.

Leroy, X. (1997) *The Caml Light System, Release 0.74, Documentation and User's Guide*. INRIA, <http://caml.inria.fr/man-caml/index.html>.

Leroy, X., Doligez, D., Garrigue, J., Remy, D. and Vouillon, J. (2001) *The Objective Caml System, Release 3.02, Documentation and User's Manual*. INRIA, <http://caml.inria.fr/ocaml/htmlman/index.html>.

Lewis, E. C., Lin, C. and Snyder, L. (1998) The implementation and evaluation of fusion and contraction in array languages. *Proceedings ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*. ACM.

Lin, C. (1996) *ZPL Language Reference Manual*. UW-CSE-TR 94-10-06, University of Washington.

Lucas, J. (2001) An array-oriented (APL) wish list. *Proceedings Array Processing Language Conference 2001*. ACM-SIGAPL.

Manjikian, N. and Abdelrahman, T. S. (1995) Array data layout for the reduction of cache conflicts. *Proc. International Conference on Parallel and Distributed Computing Systems*.

Martin-Löf, P. (1980) *Constructive Mathemeatics and Computer Programming*, pp. 153–175. Logic, Methodology and Philosophy, vol. VI. North-Holland.

Maydan, D. E. (1992) *Accurate Analysis of Array References*. PhD thesis, Stanford University.

McGraw, J. R., Skedzielewski, S. K., Allan, S. J., Oldehoeft, R. R. *et al.* (1985) SISAL*: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2*. M 146, Lawrence Livermore National Laboratory, LLNL, Livermore CA.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML*. MIT Press.

Nemeth, L. and Peyton Jones, S. (1998) A design for warm fusion. In: Clack, C., Davie, T. and Hammond, K. (eds.), *Proceedings 10th International Workshop on Implementation of Functional Languages*, pp. 381–393. University College, London.

Oldehoeft, R. R., Cann, D. C. and Allan, S. J. (1986) SISAL: Initial MIMD performance results. In: Händler, W. *et al.* (eds.), *CONPAR'86: Lecture Notes in Computer Science 237*, pp. 120–127. Springer-Verlag.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall International.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press

Plasmeijer, R. and van Eekelen, M. (1993) *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley.

Plotkin, G. (1974) Call by name, call by value, and the lambda calculus. *Theor. Comput. Sci.* **1**.

Reade, C. (1989) *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley.

Roth, G. and Kennedy, K. (1996) Dependence analysis of Fortran90 array syntax. *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications.*

Scholz, S.-B. (1998a) A case study: effects of WITH-Loop-Folding on the NAS benchmark MG in SAC. In: Hammond, K., Davie, A. J. T. and Clack, C. (eds.), *Implementation of Functional Languages (IFL'98): Lecture Notes in Computer Science 1595*, pp. 220–231. London, UK. Springer-Verlag.

Scholz, S.-B. (1998b) On defining application-specific high-level operations by means of shape-invariant programming facilities. In: Picchi, S. and Micocci, M. (eds.), *Proceedings Array Processing Language Conference 98*, pp. 40–45. ACM-SIGAPL.

Turner, D. A. (1979) A new implementation technique for applicative languages. *Software–Practice & Experience*, **9**, 31–49.

Wadler, P. and Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. *POPL'89*, pp. 60–76. ACM Press.

Wadler, P. L. (1990) Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.* **73**(2), 231–248.

Wise, D. (1985) Representing matrices as quadtrees for parallel processors. *Infor. Process. Lett.* **20**, 195–199.

Wise, D. (2000) Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In: Bode, A., Ludwig, T. and Wismüller, R. (eds.), *Euro-par 2000 Parallel Processing: Lecture Notes in Computer Science 1900*, pp. 24–33. Springer-Verlag.

Wolf, M. E. and Lam, M. S. (1991a) A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Program & Data Syst.*

Wolf, M. E. and Lam, M. S. (1991b) A data locality optimizing algorithm. *Proceedings ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 30–44.

Wolfe, M. J. (1995) *High-Performance Compilers for Parallel Computing*. Addison-Wesley.

Yi, Q., Adve, V. and Kennedy, K. (2000) Transforming loops to recursion for multi-level memory hierarchies. *Proceedings ACM SIGPLAN'00 Conference on Programming Language Design and Implementation* (*PLDI*). Vancouver, Canada. ACM Press.

Zenger, C. (1998) *Indizierte Typen*. PhD thesis, Universität Karlsruhe.

Zima, H. and Chapman, B. (1991) *Supercompilers for Parallel and Vector Computers*. Addison-Wesley.

https://doi.org/10.1017/S0956796802004458 Published online by Cambridge University Press