Single **A**ssignment **C** – Functional Programming Using Imperative Style

Sven-Bodo Scholz *

Functional Languages Implementation Workshop. Paper 21

Abstract

This paper proposes a new functional programming language called SAC (Single Assignment C) which tries to combine the best of two worlds: the efficiency and portability of `C` (or `Fortran`) and the concurrency deducable from the functional paradigm. The major objectives in the design of SAC comprise support for high performance concurrent scientific applications (number crunching), a module concept which allows for the integration of non-funtional components, and a syntax as close as possible to `C`.

1 Introduction

Functional programming languages 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 scientific applications involving complex manipulations of arrays. For this class of applications the following criteria are the most important ones for the choice of a suitable language:

- availability of primitive and compound operations on arrays,
- efficiency of compiled code,
- readability of code,
- and the integration of I/O (e.g. for a graphical representation of results).

Some popular functional languages as for example Miranda [Tur85] or ML [QRM+87] do not support arrays at all. They can only be represented as nested lists or tuples. Modifications of such array representations require rather elaborate list/tuple decompositions into individual components and the subsequent composition of new lists or tuples from these components, which usually involves several defined functions.

Functional languages which support arrays can be divided into two groups. The first group includes languages as for example NIAL [JJ93] and KIR [Klu93] which provide array operations similar to those of APL [Ive62], e.g. subarray selection, folding, rotation, transposition

*Christian–Albrechts–Universität zu Kiel, Institut für Informatik und Praktische Mathematik, Preußerstraße 1 – 9, D – 24105 Kiel Germany, EMail: sbs@informatik.uni-kiel.d400.de

etc. Many programs can be written very concisely using these primitives. In NIAL programs can be specified in dimension-independent form which allows for more general function declarations und thus better reusability. Furthermore, functional compositions of these primitives are good candidates for code optimizations which eliminate the creation of intermediate arrays. This technique is applied in KIR (called condensation [Sch86]) and is similar to Haskell's deforestation [Wad90b]. It has a well-defined mathematical basis in the mathematics of arrays [Mul88, MT94], where it is called ψ-reduction.

A common feature of these primitives is that they affect *all* elements of an array in the same way. No specifications of starts, stops and strides for array indices are therefore required. However, this may turn out to be counterproductive when it comes to specifying array manipulations which involve subarrays of specific shapes, or if different operations have to be applied to disjunct subarrays. In either case, the entire array has to be partitioned accordingly before actually doing the computation, and re-combined afterwards. This may not only aggravate program design and introduce to some extent redundant operations, but in many cases it also forecloses the application of condensation techniques.

Languages of the second group allow for a more direct manipulation of arrays. Typical examples are Haskell [HJW+92] with it's array comprehensions and SISAL [Can93] with it's FOR-loops, both of which provide iterations over pre-specified index intervals. On the one hand, loops facilitate compilation to very efficient code , on the other hand, however, it excludes condensation techniques and also the design of dimension-independent expressions.

Compiling to efficient code critically depends on the size of the semantic gap between a high-level language and the machine language. Functional frills as for example higher-order functions, partial applications, lazy evaluation, and polymorphic typing are luxuries which are hardly used in numerical applications. The use of higher-order functions and partial applications requires mechanisms for creating and resolving closures. A lazy evaluation regime is burdened with the problem of space leaks which may cause exorbitant memory demands even for toy programs. Implementing a polymorphic type system generally requires dynamic type checking, even though their overhead can be reduced to a large extend by type inference. A language tuned to the needs of numerical applications should do without these frills in order to achieve utmost run-time performance. This approach is consequently implemented in SISAL [Feo92, Can93] with the result that it easily outperforms equivalent Fortran programs in a multiprocessor environment [OCA86, Can92].

SISAL seems to be most appropriate for numerical applications, though it suffers from two major deficiencies: On the one hand it does not integrate I/O operations. On the other hand, the primitive operations on arrays are mainly restricted to boundary checks and element selection, and the loop facility requires the specification of explicit starts, stops, and strides which in a way renders all programs dimension-dependent.

Moreover, the SISAL compiler is tuned for the compilation to shared-memory systems. Existing run-time systems for distributed memory systems perform poorly since they are based on a virtual shared memory concept [HB93].

There are many attempts to integrate I/O into functional languages, such as linear types [Wad90a], and monads [Wad92a, Wad92b] in Haskell, or uniqueness typing in Clean [AP92, SBvEP93]. These approaches have in common that the programmer explicitly has to create data dependencies among consecutive I/O operations which lead to very awkward notations.

To kill all these problems with one stone, we propose to use as a suitable language a functional version of C which we call SAC (for **S**ingle **A**ssignment **C**). The advantages are twofold: we

can fall back on existing compiler technology to generate fairly efficient code for a large variety of platforms, and program notation as well as programming style remain very close to C which may enhance the acceptance of the language.

In the next section we define the subset of C taken as the basis for SAC. In section 3 we introduce a dimension-independent concept of array operations which uses a combination of loop constructs similar to ZF-expressions and a set of dimension-independent primitives. In section 4 we introduce a module concept suitable for integration into SAC. It is enhanced by a class concept as it is used in object-oriented programming. The classes uniformly extend SAC by global states, objects, and functions which, in a controlled form, may introduce side-effects.

2 A functional subset of C

C is an imperative language which provides multiple assignments, statements, pointers, a cast facility etc. all of which may cause side-effects. Nevertheless, our aim is to use a subset of C as large as possible as the kernel of SAC. This subset includes statement blocks, conditionals, loops, type and function declarations, and macros. The syntax of the SAC kernel is outlined in fig.1. A SAC program basically consists of type and function definitions and a designated *main*

$$
\begin{array}{lcl}
\mathit{Program} & \Rightarrow & [\,\mathit{TypeDef}\,]^{*}\ [\,\mathit{FunDef}\,]^{*}\ \texttt{main}\ \mathit{ExprBlock} \\
\mathit{FunDef} & \Rightarrow & \mathit{Type}\,[\ ,\ \mathit{Type}\,]^{*}\ \mathit{FunId}\ (\ [\,\mathit{ArgDef}\,]^{*}\ )\ \mathit{ExprBlock} \\
\mathit{ExprBlock} & \Rightarrow & \{\ [\,\mathit{Assign}\,]^{*}\ \mathit{RetAssign}\ \} \\
\mathit{Assign} & \Rightarrow & \mathit{Id}\,[\ ,\ \mathit{Id}\,]^{*}\ =\ \mathit{Expr}\ ; \\
 & | & \mathit{SelAssign}\ ; \\
 & | & \mathit{ForAssign}\ ; \\
\mathit{RetAssign} & \Rightarrow & \texttt{return}\ (\ \mathit{Expr}\,[\ ,\ \mathit{Expr}\,]^{*}\ )\ ; \\
\mathit{SelAssign} & \Rightarrow & \texttt{if}\ (\ \mathit{Expr}\ )\ \mathit{AssignBlock}\ [\ \texttt{else}\ \mathit{AssignBlock}] \\
\mathit{ForAssign} & \Rightarrow & \texttt{do}\ \mathit{AssignBlock}\ \texttt{while}\ (\ \mathit{Expr}\ ) \\
 & | & \texttt{while}\ (\ \mathit{Expr}\ )\ \mathit{AssignBlock} \\
 & | & \texttt{for}\ (\ \mathit{Assign}\ ;\ \mathit{Expr}\ ;\ \mathit{Assign}\ )\ \mathit{AssignBlock}
\end{array}
$$

Figure 1: Core language constructs of SAC.

function. All function definitions consist of a header, and an expression block. As in many data flow languages, an expression block may produce several return values [AD79, Nik88, Can93]. Multiple assignments within a block are permitted, i.e. an identifier may be used more than once on the left-hand side of an assignment. This does not contradict the functional paradigm since multiple assignments are equivalent to nested (non recursive) LET expressions. Thus, the scope of an identifier simply extends over the sequence of statements between two consecutive assignments to it. With this interpretation in mind, it is perfectly legitimate to call SAC a single assignment language. The Church Rosser property can nevertheless be guaranteed at

the level of entire blocks if they are taken as basic units of strictly sequential computations. A SAC compiler assures sequential execution by simply taking the SAC program as a C program.

Another conflict with the functional paradigm seems to arise from the use of conditionals in SAC (*SelAssign* in fig.1). If there is an assignment which only occurs in one branch of an IF-THEN-ELSE clause, it is not statically decidable for a successive read operation on that variable, by which assignment it is bound. For example, on the left-hand side of fig.2 it is not decidable whether the identifier A in the return statement is bound by the first or the second assignment. To fix this problem, one could simply copy the statements that, within the outer block, follow

```
{ A = 3;                          { A = 3;
  if( B ) {                         if( B ) {
    A = 42;                           A = 42;
  }                                   return(A);
  return(A);                        }
}                                   else {
                          ⟹          return(A);
                                    }
                                  }
```

Figure 2: Scoping problem with conditionals.

the IF-THEN-ELSE clause into both of its branches, as is shown on the right-hand side of fig.2. However, this step can be spared since strictly sequential execution is enforced within a block, which guarantees the determinacy of results. Thus, the C program on the left is a perfectly legitimate SAC program as well. For the same reason, we can adopt all loop constructs of C into SAC.

Other violations of the functional paradigm in C arise from the use of global variables and pointers within (mutually recursive) functions. Since we wish to exploit concurrency at the function level for non-sequential program execution the use of global variables in functions as well as the use of pointers must be outlawed in SAC, i.e. there can be no void functions. Excluding pointers has far more severe consequences, especially with respect to run-time efficiency and expressive power of SAC. As in all functional languages, data structures have to be treated conceptually as non-sharable objects subject to the orderly consumption and (re-)production by operators, even if only one or several entries have to be modified. In order to overcome these problems, the subset of C which so far has been adopted as a kernel of SAC must be extended by a suitable paradigm for more powerful operations on arrays. To this end, we adopt some concepts from the mathematics of arrays proposed in [Mul88] in combination with array comprehensions similar to those used in Haskell. Our objective is to:

- make extensive use of dimension-independent operations on entire arrays whenever this is possible without redundancies, and of dimension-independent loop constructs whenever operations must be performed on selected subsets of array elements;
- avoid, whenever possible, the generation of data structures by recursive induction over their elements;
- condensate consecutive operations on data structures by compilation techniques based on ψ-reduction principles described in [MT94] which are to avoid the creation of temporary structures, whenever possible;

- make destructive updates, whenever possible, even if this means re-organization of code similar to what is proposed in [SCA93].

Since we want to concentrate on scientific applications, the first version of Sac will only support arrays as data structures. The next section will give a detailed description of the constructs for array manipulation integrated in Sac.

3 Arrays in Sac

An array can be described as a sequence of elements and a structure imposed on it. The structure, usually called shape, defines for each dimension of the array the number of elements. A shape is represented as a vector of natural numbers whose product equals the total number of elements of the array. In Sac, the distinction between the sequence of elements and the shape of an array is made explicit. In fig.3 four different variants of array definitions are shown. In

```
{ int[ 2, 3] A = [ 1, 2, 3, 4, 5, 6];
  int[ 2, 3] B;
  int[] C = [ 1, 2, 3, 4, 5, 6];
  int[] D;

  B = [ 1, 2, 3, 4, 5, 6];
  C = reshape( C, [ 2, 3]);
  D = reshape( [ 1, 2, 3, 4, 5, 6], [ 2, 3]);
  ⋮
}
```

Figure 3: Different ways of static array definitions in Sac.

fact, all these arrays are the same: the integer numbers 1..6 are placed in an array of shape `[2, 3]`. The shape is specified either as part of a variable declaration (as done for `A` and `B` in fig.3) or by means of the primitive operation `reshape` which associates a shape with a sequence. As a consequence, all arrays can be described uniformly by flat vectors without any nestings of brackets.

The mathematics of arrays introduced in [Mul88] defines a set of dimension-independent primitive operations on arrays which are well suited as a basis for the primitives available in Sac. Let A, B denote arrays, and let $v = [v_0, ..., v_{k-1}]$ denote a vector of integers. Then

`dim` (A) returns the dimensionality of A;

`shape` (A) returns the shape vector of A;

Op (arg_1, arg_2) with $Op \in \{+, -, *, /\}$ is an extension of the respective binary operations on scalars. If one argument is a scalar and the other is an array, Op applies the scalar to each element of the array. If both arguments are arrays, provided that both have the same shape, Op is applied elementwise. As usual, these arithmetic operations can be used in infix notation as well;

`psi` (v, A) returns an subarray of A selected by the index vector v, provided that $k \leq \texttt{dim}(A)$ and $v \leq \texttt{shape}(A)$ componentwise over all indices $j \in [0, ..., k-1]$. Instead of $\texttt{psi}(v, A)$ the notation $A[v]$ can be used as well;

`take` (v, A) returns a subarray of A of shape v whose components are taken from the frontends of the respective axes, provided that $k = \texttt{dim}(A)$ and $v \leq \texttt{shape}(A)$ componentwise;

`drop` (v, A) is complementary to `take` in that we have $\texttt{drop}(v, A) = \texttt{take}(v - \texttt{shape}(A), A)$;

`reshape` (v, A) gives the array A a new shape v provided that $\prod_{l=0}^{k-1} v_l = \prod_{l=0}^{\texttt{dim}(A)} \texttt{shape}(A)[l]$. If A is specified as a scalar value, than this operation creates a new array of shape v with all elements set to this value;

`cat` (m, A, B) concatenates the arrays A and B along the axis m, provided that $1 \leq m \leq \texttt{dim}(A)$ and the shapes along the other axes are the same;

`rotate` (m, n, A) rotates the elements of A by n positions along the axis m, provided that $m \leq \texttt{dim}(A)$. The array is rotated towards increasing indices if $n \geq 0$, and towards decreasing indices otherwise.

If any of these functions is applied to a set of arguments that are not compatible in shape or type, either the type checking system or the run-time system produces an error.

In order to provide a flexible mechanism for the manipulation of subarrays, SAC includes loop primitives called WITH loops which are similar to ZF-expressions (see fig.4). They consist of

$$
\begin{array}{lll}
\mathit{ArrayExpr} & \Rightarrow & \texttt{with} \ \ (\ \mathit{Generator}\ \lceil\ ;\ \mathit{Filter}\ \rfloor^{*}\ \ )\ \ \mathit{ConExpr} \\
\mathit{Generator} & \Rightarrow & \mathit{Expr}\ <=\ \mathit{Id}\ <=\ \mathit{Expr} \\
\mathit{Filter} & \Rightarrow & \mathit{Expr} \\
\mathit{ConExpr} & \Rightarrow & \texttt{genarray}\ \ (\ \mathit{Vect}\ )\ \mathit{ExprBlock} \\
 & \mid & \texttt{modarray}\ \ (\ \mathit{Array}\ )\ \mathit{ExprBlock}
\end{array}
$$

Figure 4: Loop expressions for array manipulation in SAC.

three parts: a generator part, a filter part, and an operation part. The generator part consists of three components: two expressions which are supposed to evaluate to vectors that specify boundaries for index vectors, and a generator variable Id which stands for all the index vectors in between. The optional filter part may be used to select a subset of the indices specified in the generator part.

For array manipulations there are two kinds of operation parts: `genarray` followed by a shape vector creates a new array of the pre-specified shape, and `modarray` modifies an array. The expression block defines the value of the element of the resulting array whose position (index) is specified by the generator variable.

To illustrate the application of the different array manipulation constructs, let us consider, as a typical example for scientific applications, the numerical solution of PDEs (partial differential equations) by Gauss-Seidel relaxation [Hac93].

In a very simple form it may be defined as follows: let A^k $k \in \{1, ..., l\}$ be an array A of shape $[\underbrace{m, ..., m}_{n}]$ which results from some k iteration steps. Then the elements of A^k must be computed from those of A^{k-1} by the following algorithm:

$$A^k[i_1, ..., i_n] = \begin{cases} A^{k-1}[i_1, ..., i_n] & \text{if } \exists j \in [1, n] : (i_j = 0) \vee (i_j = m - 1) \\ \omega_1 * A^{k-1}[i_1, ..., i_n] - \omega_2 * S & \text{otherwise} \end{cases}$$

with $S = \sum_{j=1}^{n}(A^{k-1}[i_1, ..., i_{j-1}, i_j - 1, i_{j+1}, ..., i_n] + A^{k-1}[i_1, ..., i_{j-1}, i_j + 1, i_{j+1}, ..., i_n])$.

Note that only the inner elements of A are modified, whereas the boundary elements are kept unchanged.

Now, let us try to express this algorithm in SAC, using solely the primitive operations introduced above. The modification of the inner elements can easily be specified as:

```
relax( A )
...
B = omega1 / omega2 * A;
for(d=0; d<dim(A); d++) {
  B -= rotate( d, 1, A);
  B -= rotate( d, -1, A);
}
B = omega2 * B;
...
```

Rotating the argument array `A` generally brings new values into, and thus corrupts, the boundary positions, which have to be set to the old values again for the next relaxation step. All boundary values must therefore be removed from the new array, and the values of the original array must be replaced. This can be formulated in SAC as:

```
...
/* separating the "new" inner elements */
small_B = drop( reshape( dim(B), 1), take( shape(B) - 1, B))
for(d=0; d<dim(A); d++) {
  /* cutting off the left border from A in dimension d */
  drop_vect_1 = cat(0, reshape( [d+1], 0), reshape( dim(A)-d-1, 1));
  take_vect   = cat(0, cat(0, take(d, shape(A)), [1]),
                       drop( d+1, shape(small_B)) );
  left_border = take( take_vect, drop( drop_vect_1, A));

  /* cutting off the right border from A in dimension d */
  drop_vect_2 = cat(0, cat(0, reshape( [d-1], 0), shape(A)[d]),
                       reshape(dim(A)-d),1);
  right_border = take( take_vect, drop( drop_vect_2, A));

  /* concatenating small_B with the borders from A */
  small_B = cat(d, cat(d, left_border, small_B), right_border);
}
return(small_B);
}
```

This part of the relaxation algorithm requires rather tricky programming and introduces to some extent redundant operations so that all the advantages of the first part of the algorithm are neutralized.

Using a WITH-loop instead results in a more concise and irredundant program:

```
relaxDI( A )
...
A = with ( reshape( dim(A), 1) <= x <= (shape(A) - 2) )
    modarray(A) {
      tmp = omega1 / omega2 * A[x];
      for(d=0; d<dim(A); d++) {
        tmp -= rotate( d, 1, A)[x];
        tmp -= rotate( d, -1, A)[x];
      }
      return(omega2 * tmp);
return(A);
}
```

The inner FOR-loop of this program is nearly identical to the one of the first program. It iterates over all dimensions of the array, but computes only the element `A[x]` selected by the index vector `x` rather than computing the entire array. The WITH-loop repeats these iterations over the entire range of the index vectors for the inner elements of the array.

The most important feature of this program is that it is dimension-independent. We accomplish this by infering the index vectors in the generator part directly from the dimension and shape of the matrix A. In other languages, as for example in Haskell or SISAL, this can not be done since the dimensionalities of the arrays are fixed.

Being restricted to fixed dimensionalities has some unpleasant consequences. Programs must be rewritten when using them for arrays of different dimensionalities. Moreover, within one program several copies of the same algorithm (procedure) may have to be maintained in order to apply them to arrays of different dimensionalities.

4 Modules and Classes in SAC

The module concept in SAC is quite similar to that of Clean [BvEvLP87] or Haskell. It distinguishes between module declarations and module implementations. Whereas the module implementation includes the source codes for the functions and data structures, the module declaration contains the specifications necessary for its external use. The syntax of module declarations in SAC is defined in fig.5.

A module declaration consists of imports from other modules and of exports which are preceded by the keyword `own`. Export and import descriptions both consist of three (optional) parts (*EIDesc* in fig.5). First, so-called implicit types can be declared; implicit types are type names whose type definition is only known within the module; they realize the concept of abstract datatypes [Tur85, HJW+92, PvE93]. Explicit types are known to the module itself and to programs which import from this module. Finally, functions can be declared as usual, i.e. with its argument and result types.

To motivate the introduction of classes into SAC, let us consider as an example a program for a ray tracing movie. A ray tracing scene typically consists of different objects. Following

$$
\begin{array}{lcl}
ModulDec & \Rightarrow & \texttt{ModulDec}\ Id\ :\ \lceil\, ImportBlock \,\rfloor^{*}\ \texttt{own}\ :\ EIDesc \\
ImportBlock & \Rightarrow & \texttt{import}\ Id\ :\ EIDesc \\
EIDesc & \Rightarrow & \texttt{all} \\
 & | & \{\ \lceil\, ImpTypes \,\rfloor \lceil\, ExpTypes \,\rfloor \lceil\, FunDecls \,\rfloor\ \} \\
ImpTypes & \Rightarrow & \texttt{implicit types}\ :\ \lceil\, Id\ ;\, \rfloor^{+} \\
ExpTypes & \Rightarrow & \texttt{explicit types}\ :\ \lceil\, Id\ =\ Type\ ;\, \rfloor^{+} \\
FunDecls & \Rightarrow & \texttt{funs}\ :\ \lceil\, Types\ Id\ (\ Types\ )\ ;\, \rfloor^{+}
\end{array}
$$

Figure 5: Modul declarations in SAC.

established concepts of object oriented programming, a module has to be defined for each object type. It must contain functions for creating, moving, changing an object, and for its reflection properties. Fig.6 shows a typical module declaration for a glass sphere.

```
ModulDec Sphere:

own:{

  implicit types: SphereID;

  explicit types: Pos = array of int;
                  Vect = array of int;
                  Vectlist = array of vect;

  funs:
    SphereID       CreateSphere(Pos position, int radius);
    SphereID, Bool MoveSphere(SphereID Sphere, Pos new_position);
    SphereID, Bool ChangeSphere(SphereID Sphere, int new_radius);
    Bool, Vectlist RayHitsSphere(SphereID Sphere,
                                 Pos eye_position, Vect ray_direction);
  }
```

Figure 6: Modul declaration for a glass sphere in SAC .

To hide the internal representation of the glass sphere, an implicit type `SphereID` is defined. Objects can now be created, modified, and copied solely by functions provided by the module.

The module functions of the object can be divided into two groups: those that perform only read operations on, and those which modify data structures. `RayHitsSphere` belongs to the first group, whereas `MoveSphere` and `ChangeSphere` belong to the second group. In order to integrate functions that modify objects (data structures) into the functional paradigm, the data structures have to be copied whenever the objects are shared among independent program terms. In the example such an implicit copying must be outlawed since the module `Sphere` is used as a class, i.e. we expect to have explicit control over the number of objects generated. The code sequence depicted in fig.7 demonstrates how easily two spheres `A` and `S` can be created without using `CreateSphere` more than once.

Implicit copying can only be avoided by forcing the programmer to assure sequential access to the data structure. To give the programmer the free choice as to whether or not a data structure is restricted to sequential access, SAC distinguishes between modules (non-restricted)

```
...
 S = CreateSphere([100,100,100], 30);
 A, F = MoveSphere(S, [105,105,105]);
 S, F = ChangeSphere(S, 34);
...
```

Figure 7: Example code sequence using the module/class Sphere.

and classes (restricted). To do so, a uniqueness type system [AP92, SBvEP93] is implemented in Sac, and all implicit types within classes are per definition unique. Thus, substituting the keyword `ModulDec` by `ClassDec` in fig.6 leads to a typing error for the code sequence of fig.7.

Ruling out implicit copying by forcing sequential access to such an object has some consequences for programs which import such a module. Since the type of the object is hidden and no implicit copies can be made any more, for these programs objects become pure handles (pointers) for global objects. Once a global (sphere) object is created, it is not necessary to always get back the pointer to the sphere as a result of a modification operation. On the contrary, returning a pointer to the sphere insinuates that it might have changed, which indeed is not true since the pointer always remains the same. To improve the readability of code, these return values can be omitted in Sac. By doing so, side-effects on the `SphereID` are introduced by the functions `MoveSphere` and `ChangeSphere`. The respective class declaration is given in fig.8.

```
ClassDec Sphere:

own:{

  implicit types: SphereID;

  explicit types: Pos = array of int;
                  Vect = array of int;
                  Vectlist = array of vect;

  funs:
    SphereID       CreateSphere(Pos position, int radius);
    Bool           MoveSphere(SphereID Sphere, Pos new_position);
    Bool           ChangeSphere(SphereID Sphere, int new_radius);
    Bool, Vectlist RayHitsSphere(SphereID Sphere,
                                 Pos eye_position, Vect ray_direction);
  }
```

Figure 8: "Imperative" class declaration for a sphere in Sac .

However, these side-effects do not violate the Church Rosser property as long as sequential access to the data structure `SphereID` is guarantied. In order to be able to assure sequential access by a uniqueness type system, the omitted return values have to be re-introduced by the compiler as an intermediate step of compilation.

The example shows that introducing objects and classes is equivalent to introducing global objects and explicit handles (pointers) to them. Being able to handle global objects without

implicit copying is exactly what is needed to do I/O. Therefore, the classes of Sac are an elegant way for integrating I/O. Moreover, separating I/O by the introduction of classes buys the following advantages for free:

- easy integration of I/O libraries written in other languages than Sac ,
- non-monolithic I/O; i.e. I/O on objects that do not interfere can be done concurrently,
- and I/O operations are conceptually separated from the otherwise strictly functional code by encapsulating them in classes.

5 Conclusion

In this paper, we present the basic concepts of a new functional programming language Sac. The key motivation for the development of Sac derives from the observation that most functional languages are not very suitable for scientific applications. Either the support of functional frills severely limits the run-time performance, or a restricted syntax forces the programmer to write dimension-dependent code, or to implement I/O-interfaces in other (non functional) programming languages. With Sac we try to provide a functional language which supports dimension-independent array operations as well as a sophisticated module/class concept. This class concept integrates the basic mechanisms of object oriented programming (with desired side-effects) cleanly into the functional paradigm and thus allows to use I/O packages and procedures coded in other languages than Sac. Furthermore, being closely related to C, Sac enables C programmers to easily write functional programs whose inherent concurrency can be exploited by the compiler.

References

[AD79] W.B. Ackerman and J.B. Dennis: *VAL-A Value-Oriented Algorithmic Language: Preliminary Reference Manual*. TR 218, MIT, Cambridge, MA, 1979.

[AP92] P. Achten and R. Plasmeijer: *The Beauty and the Beast*. University of Nijmegen, 1992.

[BvEvLP87] T.H. Brus, M.C. van Eekelen, M.O. van Leer, and M.J. Plasmeijer: *CLEAN: A Language for Functional Graph Rewriting*. In G. Kahn (Ed.): FPCA'87, LNCS, Vol. 274. Springer, 1987.

[Can92] D.C. Cann: *Retire Fortran? A Debate Rekindled*. Communications of the ACM, Vol. 35(8), 1992, pp. 81–89.

[Can93] D.C. Cann: *The Optimizing SISAL Compiler: Version 12.0*. Lawrence Livermore National Laboratory, LLNL, Livermore California, 1993. part of the SISAL distribution.

[Feo92] J.T. Feo: *SISAL*. Technical Report UCRL-JC-110915, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1992.

[Hac93] W. Hackbusch: *Iterative Lösung großer schwachbesetzter Gleichungssysteme.* Teubner Studienbücher Mathematik. Teubner, 1993.

[HB93] M. Haines and W. Böhm: *Task Management, Virtual Shared Memory, and Multithreading in a Distributed Memory Implementation of SISAL.* In A. Bode et al. (Eds.): PARLE '93, LNCS, Vol. 694. Springer, 1993, pp. 12–23.

[HJW+92] P. Hudak, S. Peyton Jones, P. Wadler, et al.: *Report on the Programming Language Haskell.* Yale University, 1992.

[Ive62] K.E. Iverson: *A Programming Language.* Wiley, New York, 1962.

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

[Klu93] W.E. Kluge: *A User's Guide for the Reduction System.* Internal Report, University of Kiel, 1993.

[MT94] L. Mullin and S. Thibault: *A Reduction Semantics for Array Expressions: The PSI Compiler.* Technical Report CSC-94-05, University of Missouri-Rolla, 1994.

[Mul88] L.M. Restifo Mullin: *A Mathematics of Arrays.* PhD thesis, Syracuse University, 1988.

[Nik88] R.S. Nikhil: *ID Version 88.1, Reference Manual.* CSG Memo 284, MIT, Laboratory for Computer Science, Cambridge, MA, 1988.

[OCA86] R.R. Oldehoeft, D.C. Cann, and S.J. Allan: *SISAL: Initial MIMD Performance Results.* In W. Händler et al. (Eds.): CONPAR '86, LNCS, Vol. 237. Springer, 1986, pp. 120–127.

[PvE93] R. Plasmeijer and M. van Eekelen: *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[QRM+87] D. Mac Queen, R.Harper, R. Milner, et al.: *Functional Programming in ML.* Lfcs education, University of Edinburgh, 1987.

[SBvEP93] S. Smetsers, E. Barendsen, M. van Eeklen, and R. Plasmeijer: *Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs.* Technical report, University of Nijmegen, 1993.

[SCA93] A.V.S. Sastry, W. Clinger, and Z. Ariola: *Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates.* In FPCA '93, Copenhagen. ACM, 1993, pp. 266–275.

[Sch86] C. Schmittgen: *Spezifikation der Architektur und Realisierung eines Reduktionssystems mit konsequenter Unterstützung strukturierter Datenobjekte und n-stelliger definierter Funktionen.* PhD thesis, TU Berlin, 1986.

[Tur85] D.A. Turner: *Miranda: a Non-Strict Functional Language with Polymorphic Types.* In IFIP '85, Nancy, LNCS, Vol. 201. Springer, 1985.

[Wad90a] Philip Wadler: *Linear types can change the world!* In M. Broy and C.B. Jones (Eds.): Programming Concepts and Methods. Noth Holland, 1990.

[Wad90b] P.L. Wadler: *Deforestation: transforming programs to eliminate trees.* Theoretical Computer Science, Vol. 73, 1990, pp. 231–248.

[Wad92a] P. Wadler: *Comprehending Monads.* Mathematical Structures in Computer Science, 1992.

[Wad92b] P. Wadler: *The essence of functional programming.* In POPL '92, Albequerque, 1992.