On Interfacing SAC Modules with C Programs (Draft paper)

Nico Marcussen-Wulff and Sven-Bodo Scholz

Dept of Computer Science, University of Kiel, 24118 Kiel, Germany
e-mail: {nmw,sbs}@informatik.uni-kiel.de

This paper is on interfacing *C* programs with modules written in SAC, a functional language with powerful array processing facilities [?].

Interfacing program modules written in different languages has become an increasingly important issue of economic software design. It allows to re-use as much as possible existing code (program libraries) and to import special language features not available in the language in which the main program is written.

Language interfaces usually include two complementary parts, one to import specifications from, the other to export specifications to foreign languages. Straightforward designs try to map data types as well as function signatures more or less directly from one language to another, i.e., with as few modifications as possible. This approach works fairly well if the features of the imported language have direct counterparts in the inporting language. Examples in the functional world are GreenCard for HASKELL[?], the foreign language interface of SISAL [?], or the integration of *C* libraries into SAC.

However, things become decidedly more complicated if the imported language supports more powerful features than the importing language, such as, e.g., polymorphic typing, type classes, or generic functions, which is one of the primary reasons of language interfacing in the first place. Here the interfaces have to provide the mechanisms necessary to map the constructs of the imported language to some 'work-around' implementations in the importing language, which usually requires some program annotations to guide the code generation for the interface. However, using such interfaces often requires some intimate knowledge of the runtime system of at least one of the two languages involved, and mimiking features of the imported language may spoil an otherwise clean design of the importing language. An example in kind is the *C* interface of HASKELL [?,?]. Since *C* does support neither polymorphism nor overloading, all instances of polymorphic | overloaded HASKELL functions called by a *C* program must be explicitely exported through the interface.

The problems of designing an interface through which SAC modules may be imported into *C* programs are of a similar nature. As most other imperative languages, *C* gives only minimal support for arrays; dimensionalities and shapes must be explicitely declared, and direct operations on arrays are basically confined to indexing individual array elements, i.e., programming complex numerical applications becomes a rather loop-intensive, tedious, time-consuming and error-prone undertaking, resulting in fairly complex, dimension- and shape-specific code which can hardly be used in contexts other than the one for which it was developed.

Importing Sac modules into *C* programs may help to overcome a great deal of these problems, without asking the programmer to switch to much of another syntax. Sac is a functional language based on *C* syntax which supports a powerful array processing concept. Very much like Apl, it allows to specify generic (and overloaded) functions applicable to arrays of varying dimensionalities and shapes. The Sac compiler, in conjunction with a sophisticated type inference subsystem, generates highly efficient dedicated code for instances of these functions which operate on arrays of specific dimensionalities and shapes, as they can be inferred from actual argument arrays. Moreover, the Sac compiler includes facilities to generate highly optimized concurrently executable code for shared-memory multi-processor systems.

Unlike other foreign language interfaces, the Sac$\rightarrow$ *C* interface does not try to directly map Sac types into *C* types but instead uses in *C* the notion of abstract Sac expression types. This abstraction has several advantages for both application programmers and language implementors. Hiding the actual implementation of structured data such as arrays from the user gives the compiler the freedom to internally choose among several representations. Also, representations can be changed without affecting the interface and thus existing program specifications. Last but not least, generic Sac functions can be directly called from the *C* program. They are implemented as wrapper functions which include optimized codes for all specialized function instances, i.e., for all parameter types that may actually occur. The wrappers handle the dynamic dispatches to the appropriate codes based on actual function parameters.

As an illustration of how Sac modules can be interfaced with *C* programs, consider the `main()` function of fig. 1. It calls upon two Sac functions `addValue( .. )` and `addVectors( .. )`, specified in a file `SACMod` shown in fig. 2 and imported through the header file `SACMod.h`. They respectively add to all elements of an array the same parameter value and perform an elementwise addition on two arrays of compatible shapes. These function calls are to add to a *C* vector `int_data_in` of some 20 elements (generated by a `for` loop) some constant value of 99, add the vector thus obtained elementwise to itself, and return the resulting vector `int_data_out` as output. To pass vectors (and an integer parameter) between Sac and *C* , the `main` program introduces variables `vector_in`, `vector_out` and `par` of the abstract data type `SAC_arg`, and uses standard conversion functions `SAC_Intarray2Sac`, `SAC_Int2Sac` and `SAC_2Intarray` to convert the *C* vector `int_data_in` into the Sac vector `vector_in`, the constant 99 into the Sac parameter `par`, and the Sac vector `vector_out` back into the *C* vector `int_data_out`, respectively. The prototypes of these functions are held in the header file `sac_cinterface.h`, which in fact defines the interface.

The functions defined in the module `SACMod` (compare fig. 2) `addValue` adds, by means of the overloaded operator +, the value substituted for the parameter `value` to all elements of an integer array of any dimensionality and shape, and `addVectors` adds, again by means of the overloaded operator +, elementwise two integer arrays (substituted for the parameters `v1` and `v2`) of identical but otherwise freely chosen dimensionalities and shapes. However, for the Sac com-

```
/* *******  C program using a SAC module exported as a C library ****** */
#include <stdlib.h>
#include <stdio.h>
#include "sac_cinterface.h"
#include "SACMod.h"

#define SMALLARRAY 20

int main()
{
  int i;
  int *int_data_in;
  int *int_data_out;

  SAC_arg vector_out;
  SAC_arg vector_in;
  SAC_arg par;

  SAC_InitRuntimeSystem();

  /* init data */
  int_data_in=(int*)malloc(SMALLARRAY*sizeof(int));
  for(i=0; i<SMALLARRAY; i++) {int_data_in[i]=i+1;}

  /* convert C data to abstract datatype SAC_arg */
  vector_in = SAC_IntArray2Sac(SAC_CONSUME_ARG, int_data_in, 1, SMALLARRAY);
  par = SAC_Int2Sac(99);

  /* call SAC-function addValue */
  if(SAC_SACMod_addValue_1_2(&vector_out, vector_in, parameter)) {
    printf("some error occurred, calling the function!\n\n");
    exit(1);
  }

  /* use result of previous for next function call */
  vector_in = vector_out;

  /* increase refcounter to use argument twice */
  SAC_SetRefcounter(vector_in, 2);

  /* call SAC-function addVectors */
  if(SAC_SACMod_addVectors_1_2(&vector_out, vector_in, vector_in)) {
    printf("some error occured, calling the function!\n\n");
    exit(1);
  }

  /* convert and consume the result */
  int_data_out = SAC_Sac2IntArray(SAC_CONSUME_ARG, vector_out);

  /* print result */
  /* print elements of C array */
  for(i=0; i<SMALLARRAY; i++)
    printf("%d ", int_data_out[i]);
    printf("\n");

  free(int_data_out);
  SAC_FreeRuntimeSystem();
  return(0);
}
```

Fig. 1. A *C* program interfacing with a SAC module

piler to be able to generate thoroughly optimized code, the `SACmod` module must be complemented by an additional specialization file `ModuleSpec` which specifies the concrete function instances that are called from the *C* program (see the fig. 3).

```
/* SAC module of generic function definitions */

Module SACMod:

/* add value to all elements of an array/vector */
int[] addValue(int[] v, int value)
{
  return(v+value);
}

/* add two arrays/vectors */
int[] addVectors(int[] v1, int[] v2)
{
  return(v1+v2);
}
```

Fig. 2. SAC module for the SAC functions used in the program of fig. 1

The compiler takes both specifications to generate the the C-library libSACMod.a and the header file `SACMod.h` shown in fig. 4 which in fact constitutes the interface for the user defined SAC functions. Here the function names are prefixed with `SAC_SACMod` to identify them as SAC functions imported from the module SACMod and postfixed with _1_2 to indicate that they take two arguments and produce one result value.

The codes for all specializations of SAC functions with the same name and numbers of input / output parameters (SAC functions may have more than one of the latter) are encapsulated in a wrapper function featuring the same header as the one found in the SACMod.h file. It analyses the actual input parameters with which the function is called through the interface, selects for execution the matching specialized code, and throws an error message if there is no match.

Another problem that must be taken care of at the SAC$\rightarrow$ *C* interface concerns the reference counting mechanism which enables the SAC runtime system to release as early as possible heap space occupied by SAC data structures that are no longer referenced. Following the functional paradigm, SAC functions conceptually consume their arguments, thus decrementing their reference counts by one, and produce one result value each which may subsequently have to be copied as many times as required for consumption by function calls elsewhere, incrementing their reference counts accordingly.

```
/* specializing the generic functions for the following special shapes */
ModuleSpec SACMod :

own:
{
functions:
  int[] addValue(int[10] v, int value);
  int[] addValue(int[20] v, int value);
  int[] addValue(int[30] v, int value);

  int[] addVectors(int[10] v1, int[10] v2);
  int[] addVectors(int[20] v1, int[20] v2);
  int[] addVectors(int[30] v1, int[30] v2);
}
```

Fig. 3. Specialization file for the SAC functions used in the program of fig. 1

```
/* *******  headerfile SACMod.h ****** */
/* Interface SAC <-> C for SACMod
 */

#include "sac_cinterface.h"

/* function addValue
 * defined in module SACMod
 * accepts arguments as follows:
 * int[10] SACl_v, int SACl_value -> int[10] out1
 * int[20] SACl_v, int SACl_value -> int[20] out1
 * int[30] SACl_v, int SACl_value -> int[30] out1
 */
extern int SAC_SACMod_addValue_1_2(SAC_arg *out1, SAC_arg in1, SAC_arg in2);


/* function addVectors
 * defined in module SACMod
 * accepts arguments as follows:
 * int[10] SACl_v1, int[10] SACl_v2 -> int[10] out1
 * int[20] SACl_v1, int[20] SACl_v2 -> int[20] out1
 * int[30] SACl_v1, int[30] SACl_v2 -> int[30] out1
 */
extern int SAC_SACMod_addVectors_1_2(SAC_arg *out1, SAC_arg in1, SAC_arg in2);
```

Fig. 4. The header file `SACMod.h`

When dealing with Sac structures at the interface, some of the operations on reference counts have to be explicitely specified by the application programmer, using standardized functions imported into the *C* world through the `sac_cinterface.h` header file. A typical example in the program of fig. 1 concerns the duplication of the input parameter `vector_in` in the application of `addVectors`. This parameter being instantiated with a (reference to) a Sac array, its reference count must be explicitely set to the value 2 by the function `SAC_SetRefcounter` before calling `addVector` since the results of all imported SAC-functions (including conversion functions) by convention have an inital reference count of 1. In order to keep the overhead of the interface low, the array conversion functions `SAC_IntArray2Sac` and `SAC_Sac2IntArray` also include a flag parameter `SAC_Consume_Arg` which indicates whether their array arguments are just being consumed or whether further copies are used elsewhere in the *C* program.

Other support functions provided by the `sac_cinterface.h` file include the functions `SAC_InitRuntimeSystem()` and `SAC_FreeRuntimeSystem()` (see fig. 1) which respectively start the Sac runtime environment before calling the Sac interface and clean up whatever is left of it after having completed all Sac computations.

References

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

[FLMJ98] S. Finne, D. Leijen, E. Meijer, and S.L. Peyton Jones. H/Direct: A Binary Foreign Language Interface for Haskell. In *Proccedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98), Baltimore, Maryland, USA*, volume 34.1 of *ACM SIGPLAN Notices*, pages 153–162. ACM Press, 1998.

[FLMJ99] S. Finne, D. Leijen, E. Meijer, and S.L. Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99), Paris, France*, volume 34.9 of *ACM Sigplan Notices*, pages 114–125. ACM Press, 1999.

[JNR97] S.L. Peyton Jones, T. Nordin, and A. Reid. Green Card: a Foreign-Language Interface for Haskell. In *Proceedings of the Haskell Workshop, Amsterdam, The Netherlands*, 1997.

[Sch98] S.-B. Scholz. On Defining Application-Specific High-Level Array Operations by Means of Shape-Invariant Programming Facilities. In S. Picchi and M. Micocci, editors, *Proceedings of the International Conference on Array Processing Languages (APL'98), Rome, Italy*, pages 40–45. ACM Press, 1998.