Classes and Objects as Basis for I/O in SAC

Clemens Grelck and Sven-Bodo Scholz *

November 12, 1998

Abstract

In imperative languages I/O is realized through sequences of side-effecting function applications/ procedure invocations. This seems to be a suitable way of specifying I/O since it coincides with an intuitive understanding of it as sequences of actions. In contrast, functional languages carefully have to avoid side-effects to sustain referential transparency. Many different solutions, such as dialogues, continuations, monads and uniqueness typing have been proposed.

The I/O facilities of SAC are based on uniqueness typing. Instead of using an explicit type attribute as in CLEAN, unique types are introduced as special modules called *classes*. To provide a syntax as close as possible to that of imperative languages, we propose two new mechanisms to be included on top of classes in SAC: a *call-by-reference mechanism* and *global objects*. Although a combination of both mechanisms allows for very imperative-like notations, we can define a purely functional semantics. Thus we combine the advantages of referential transparency with the expressiveness of imperative I/O. Moreover, these two mechanisms allow the programmer to introduce and manipulate states of arbitrary data-structures within SAC.

1 Introduction

SAC[Sch94] is a programming language primarily intended for numerical applications. For the efficient implementation of large numerical applications on multiprocessor architectures extensions of imperative languages (HPF[For94], C*[Fra91], etc.) as well as tailor-made functional languages (SISAL[Can93], etc.) have been proposed. The most important advantage of the functional over the imperative paradigm is its referential transparency, or the absence of side-effects. All function applications can be treated as values, i.e., many well-known compiler optimizations such as common subexpression elimination, loop invariant removal, and others can be generalized to eliminate superfluous function applications as well. The Church-Rosser-Property guarantees the

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

determinacy of results irrespective of execution orders, facilitates the identification of program parts that can be evaluated concurrently, and thus compilation to multiprocessor architectures.

Though numerical applications written in the functional language SISAL[Can93] have demonstrated their superiority over equivalent FORTRAN programs, easily outperforming them by factors of more than two in multiprocessor systems, there are still considerable acceptance problems. Apart from pragmatic considerations pertaining to investments made in a large body of programs written in imperative languages which cannot easily be disposed of, there is a considerable reluctance to familiarize with the functional programming styles and with some syntactical peculiarities of functional languages.

In order to overcome these acceptance problems, we take the alternative approach of identifying in the widely accepted imperative language C a subset whose semantics allows for a purely functional interpretation. This language, to which we will refer as SAC (for Single Assignment C) to emphasize its functional character, also allows us to fall back on highly sophisticated existing compiler technology. Most of the basic constructs of C can be adopted as a kernel of SAC (for details see [Sch94]):

- *Multiple Assignments* can be viewed as nested single assignment blocks, i.e. the scope of a variable simply extends over the sequence of statements between two consecutive assignments to it.

- *C-conditionals*, which apparently violate the static scoping rule, are integrated into the functional world by copying the assignments that follow a conditional into both of its branches.

- *C-loop-constructs* are treated as shortcut notations for tail-end recursive functions similar to the loop-constructs available in single assignment languages like ID[AGP78], VAL[AD79], or SISAL[Can93].

It is well accepted that functional languages need to include mechanisms which allow for an orderly interaction with a state. However, it must be kept free of side-effects in order to sustain the Church-Rosser-Property. These mechanisms are primarily required for I/O purposes and for updating operations on large data-structures, which in a purely functional world would otherwise have to be copied several times. The integration of explicit states and destructive updates is also of interest from a software engineering point of view: when structuring a program by the data-structures it operates on, i.e. according to the object-oriented paradigm, it is crucial for the programmer to have explicit control over the number of objects (data-structures) generated, over each object's lifetime, and over modification facilities.

Several solutions for the integration of states and destructive updates into the functional paradigm have been proposed (for surveys see [HS89, Per91, JW93, AP95, CH93]), which may roughly be classified as stream-based approaches and environment-based approaches.

In stream-based approaches the communication with the outer world is done via streams so that the state and the state modifications can be situated outside

the functional framework, e.g. in the operating system. These approaches are not applicable to SAC since it neither provides streams nor other conceptually infinite data-structures.

The environment-based approaches incorporate states and state-modifications into the functional world by the introduction of new objects which can only be used in a very restricted way. A state is passed as argument to a modification function either explicitly, as for instance in the uniqueness typing approach[AP95, SBvEP93], or implicitly, as for instance by a higher-order function ("bind") in the monadic approach[Wad92, JW93].

The monadic approach too is not suitable for an integration into SAC since it requires higher-order functions in conjunction with a polymorphic type system, both of which are not available in SAC. Moreover, concurrent state modifications on disjunct state-partitions (non-monolithic states) still seem to be an unsolved problem with monads[LJ94].

The uniqueness typing approach, as proposed for CLEAN[AP95], seems to be the best solution for SAC. In CLEAN, states can be represented by variables, and the modification functions, e.g. the I/O primitives, are explicitly applied to states. As a consequence, these states would have to be copied whenever they are used in two or more independent subexpressions. These situations may be avoided by means of the type attribute *uniqueness*[SBvEP93]. An expression is said to be *unique* if it is not referenced more than once in a function's body or, more precisely, in each branch of a conditional/case construct if it does not appear anywhere else in the body. So, if in all applications of a given function `f` the i^{th} argument $\mathtt{a}_i$ is unique, the heap-space holding the value of $\mathtt{a}_i$ may be re-used in the body of `f` (destructive update). In particular, a function that returns an object of the same type as one of its unique arguments may simply overwrite the heap-location of that argument.

The advantages of this approach are twofold. First, an integration into SAC requires only a small extension of the existing type-system. Second, non-monolithic state modifications can be deduced directly since all state partitions are explicitly represented by distinct variables.

However, the mechanisms for inspecting and modifying states as provided by the uniqueness typing approach have some major deficiencies:

- Modifications of a state `a` by a function `f` may only be specified as `a = f(a);` . This notation has two drawbacks: it permits renaming the state by writing `b = f(a);` instead, and it requires a return value to be specified which from a pragmatical point of view is superfluous.

- All states that need to be inspected or modified within a particular nesting of function invocations have to be explicitly passed as arguments to all intermediate functions. As a consequence, the argument and return lists of the function definitions must be extended by the state variables that have to be received from and returned to the calling functions even if none of the intermediate functions perform operations on them.

- The complete external environment is represented by a generic state "'World"'. Whenever parts of the environment need to be accessed, they

have to be explicitly extracted from and re-integrated into that state.

We found out that these deficiencies can be overcome by introducing two new mechanisms on top of uniqueness typing. The first is a *call-by-reference mechanism* for specifying state modifications without the necessity of returning the modified state. The other mechanism introduces *global objects* that can be accessed in the body of any function without passing them as arguments. By introducing two meaning-preserving transformation schemes which successively eliminate these constructs, we show that they do not violate the referential transparency. This is done by inserting additional arguments and return values as needed.

These transformations are applied before doing the compilation phases that need to exploit referential transparency, such as the above-mentioned optimizations and the partitioning of programs into concurrently executable parts. When sequential threads of C-code are generated as target code, all parameters that have been added are eliminated again.

The paper is organized as follows: chapter 2 explains how uniqueness typing and a strict separation of unique from non-unique components is achieved in SAC. These basic constructs are successively extended by the call-by-reference mechanism described in chapter 3 and by the global objects described in chapter 4. The paper concludes with a short summary given in chapter 5.

2 Uniqueness Types in SAC

Uniqueness types have proved to be a suitable mechanism for dealing with states and state transitions in a functional language [AP95, SBvEP93]. The introduction of uniqueness types requires their clean separation from regular types. In contrast to the type attribute *UNQ* of CLEAN [PvE95], SAC adopts for this purpose terms and mechanisms from the object-oriented programming paradigm. This is motivated by the observation that objects in this paradigm typically represent states, which may be created, modified, and removed under the explicit control of the application programmer.

Uniqueness types are introduced into SAC in the form of classes. In object-oriented programming, a class consists of a basic type definition and a set of functions, which exclusively grant access to data structures of the basic type. Expressions of this type are called objects of the respective class. More advanced object-oriented mechanisms, e.g. inheritance, are not required for our approach. However, their integration into SAC may be a subject of future research.

SAC already provides a suitable concept for modeling classes: the module system [Sch94]. A SAC-module consists of two files containing a module declaration and a module implementation. The module implementation comprises type and function definitions and may be specified in SAC as well as in any other programming language that is compatible with the C linkage system. The module declaration serves as an interface for the external usage of a module's types and functions. Functions are exported by declaring their names and signatures. Types may either be exported explicitly or implicitly. For an explicit type, the name as well as the implementation are given in the module declaration. In

contrast, the implementation of an implicit type is hidden within the respective module implementation and only its name is made public in the declaration. As a consequence, exported functions of the respective module remain as the sole interface through which variables of an implicit type may be accessed. Since this conforms to the specification of classes as given above, they are introduced into SAC as special modules.

Apart from using key-words like `ClassDec` instead of `ModuleDec`, the only difference between classes and modules arises from the fact that a class always provides a special implicit type. The name of this type is identical to the name of the class itself. By definition, it is a uniqueness type representing the basic type of the class. With respect to its special characteristics, the type is not mentioned in the class declaration in order to distinguish it from ordinary implicit types.

To illustrate the concepts proposed in this paper, the integration of I/O facilities into SAC is presented in the form of a case study throughout the remaining sections. Figure 1 shows a first solution based on classes and objects. The case study comprises an excerpt of the declaration of a class `File` and a small program illustrating the usage of `File`. This class is designed to permit access to the file system and the standard I/O channels.

The functions of our class `File`, as shown in the declaration of figure 1, are typical representatives of three types of class functions. A constructor function like `open_stdout` produces an object of the respective class. Objects are consumed by the application of a destructor function like `close_stdout`. The function `fprintf` represents the third type of class functions. Conceptually, it consumes a `File`-object (e.g. `stdout`) and produces a new `File`-object, but from a pragmatical point of view, this may be interpreted as the implicit modification of a single object. In this case, the interpretation reflects a possible implementation that takes advantage of the uniqueness property of objects by performing a destructive update on the respective data structure. Giving the conceptually new object the same name as the one that has been consumed underlines this idea.

A generic class named `World` is designed to allow the specification of programs which interact with their external environment. The class `World` has exactly one generic object called `world`. When given to a program's `main` function as a parameter, `world` represents the entire external environment of the program.

To allow for non-monolithic I/O, more specialized objects can be derived from `world`, which represent disjoint partitions of the environment. The application of the constructor function `open_stdout` to `world` in figure 1 extracts the standard output channel from the initially monolithic representation of the environment. This results in a new object `stdout` representing the standard output channel, and in a modified object `world` representing the remaining environment.

Once the object `stdout` is available, `world` is no longer needed to send output to the standard output channel by applications of the function `fprintf`

```
ClassDec File :
own:
{
funs: File, World open_stdout(World);
      World close_stdout(File, World);
      File fprintf(File, string);
}


import File: all;

File printline(File stdout, string message)
{
  stdout = fprintf(stdout, message);
  stdout = fprintf(stdout, "\n");
  return(stdout);
}

File Header(File stdout)
{
  stdout = printline(stdout, "This is SAC !");
  return(stdout);
}

int, World main(World world)
{
  stdout, world = open_stdout(world);
  stdout        = Header(stdout);
  stdout        = fprintf(stdout, "Hello World.\n");
  world         = close_stdout(stdout, world);
  return(0, world);
}
```

Figure 1: Providing I/O facilities based on classes.

[1].

Since interactions with the environment are part of a program's result, the representation of the modified environment has to be returned by the program's `main` function. In the example shown in figure 1, the standard output channel is re-integrated into the remaining environment by an application of the destructor function `close_stdout`. Afterwards, `world` is returned by `main` as the monolithic representation of the modified environment.

[1] In general, I/O-primitives in SAC provide a functionality similar to their counterparts in C, combining a format string with a variable number of arguments. However, some simplifications apply within this paper in order to avoid confusion.

3 A Call-by-Reference Mechanism

The only way to specify the modification of an object is by means of a function that conceptually consumes an object and produces a new one, written in the purely functional form `a = f(a);` . Since, in these cases, the explicit specification of a return value from a pragmatical point of view is superfluous, we introduce a short cut notation similar to those of call-by-reference mechanisms in imperative languages. Reference parameters are distinguished from ordinary parameters by the address operator "&"; it indicates that the respective object is returned implicitly, i.e. an explicit return value is omitted in the function's definition as well as in its applications. In order to allow the specification of functions without any explicit return value, the syntax of SAC is extended by void-functions as known from C . Figure 2 illustrates the usage of the new notation with a modified version of the case study.

We can observe that `Header` and `printline` have become void-functions. Only the address operators "&" in their parameter lists still indicate that they actually modify `stdout`. Applying the call-by-reference notation to I/O primitives like `fprintf` allows for specifying I/O operations in a way that is much more similar to C than the first approach in figure 1. This can be noticed particularly in the body of `printline`. Furthermore, `world` no longer has to be returned explicitly by `main`.

In a pre-processing step, the SAC-compiler transforms all occurrences of the short cut notation into their equivalent purely functional forms. Therefore, compilation phases that rely on referential transparency are not affected. An outline of the transformation algorithm consisting of four rules is given in figure 3.

Rules 1 and 2 make implicit return values explicit. Syntactically, this requires the introduction of additional return values and, in the case of a function definition, the extension or the reintroduction of the `return`-statement.
Example:

```
void printline(File &stdout)
{ ... }
```

is transformed into

```
File printline(File stdout)
{ ...
  return(stdout);
}
```

Rule 4 eliminates the short cut notation for function applications.
Example:

```
fprintf(stdout, message);
```

is transformed into

```
stdout = fprintf(stdout, message);
```

Unfortunately, this is not always as easy as in the above example. Problems arise from function applications that are nested within expressions, e.g. other function applications. Rule 3 takes care of this special case. Assume the following example:

```
ClassDec File :
own:
{
funs: File open_stdout(World &);
      void close_stdout(File, World &);
      void fprintf(File &, string);
}


import File: all;

void printline(File &stdout, string message)
{
  fprintf(stdout, message);
  fprintf(stdout, "\n");
}

void Header(File &stdout)
{
  printline(stdout, "This is SAC !");
}

int main(World &world)
{
  stdout = open_stdout(world);
  Header(stdout);
  fprintf(stdout, "Hello World.\n");
  close_stdout(stdout, world);
  return(0);
}
```

Figure 2: Providing I/O facilities with the call-by-reference mechanism.

```
{ A = create_stack_object();
  B = create_stack_object();
   ...
  push(B, pop(A));
   ...
  remove_stack_object(A);
  remove_stack_object(B);
   ... }
```

Let the signatures of `push` and `pop` be as follows:
`void push(stack_class&, int)` ,
`int pop(stack_class&)` .
Eliminating the call-by-reference notation according to rule 1 adds additional

1. For each function in an imported class or module declaration:
   - For each call-by-reference parameter `<type> &` :
     - Delete the address operator `&` .
     - Add `<type>` to list of return types.
2. For each function definition:
   - For each call-by-reference parameter `<type> & <ident>`:
     - Delete the address operator `&` .
     - Add `<type>` to list of return types.
     - Add `<ident>` to list of return values in the function's `return`-instruction.
3. For each function application of the form $\mathtt{r_1}, \ldots, \mathtt{r_m} = \mathtt{fun}(\mathtt{e_1}, \ldots, \mathtt{e_n});$ where $\mathtt{e_i} = \mathtt{fun2}(\mathtt{a_1}, \ldots, \mathtt{a_k})$ and each $\mathtt{e_j}, 1 \leq \mathtt{j} < \mathtt{i}$, does not contain another function application:
   - Let $\mathtt{tmp_x}$ be a new, previously unused variable of the same type as $\mathtt{e_i}$.
   - Replace $\mathtt{r_1}, \ldots, \mathtt{r_m} = \mathtt{fun}(\mathtt{e_1}, \ldots, \mathtt{e_n});$ by
     $\mathtt{tmp_x} = \mathtt{fun2}(\mathtt{a_1}, \ldots, \mathtt{a_k});$
     $\mathtt{r_1}, \ldots, \mathtt{r_m} = \mathtt{fun}(\mathtt{e_1}, \ldots, \mathtt{e_{i-1}}, \mathtt{tmp_x}, \mathtt{e_{i+1}}, \ldots, \mathtt{e_n});$
4. For each function application of the form $\mathtt{r_1}, \ldots, \mathtt{r_m} = \mathtt{fun}(\mathtt{e_1}, \ldots, \mathtt{e_n});$ where all $\mathtt{e_i}$ do not contain other function applications:
   - For each $\mathtt{e_i}$ passed using call-by-reference:
     - Extend the return list by $\mathtt{e_i}$ resulting in
       $\mathtt{r_1}, \ldots, \mathtt{r_m}, \mathtt{e_i} = \mathtt{fun}(\mathtt{e_1}, \ldots, \mathtt{e_n});$.

Figure 3: Transformation algorithm for the call-by-reference mechanism.

return values to `push` and `pop`, resulting in the modified signatures:

```
stack_class push(stack_class, int) ,
int, stack_class pop(stack_class) .
```

In the example, the short cut notation for the application of `pop` cannot be eliminated in its specific syntactical position because, in SAC, functions that are nested within other expressions are required to have exactly one return value. As a consequence, the application of `pop` must be extracted from the application of `push`. According to rule 3, the above example is transformed into:

```
{  ...
  tmp1 = pop(A);
```

```
  push(B, tmp1);
   ...  }
```

Now, the applications of `push` and `pop` can simply be adjusted to their modified signatures according to rule 4, resulting in:

```
{  ...
   tmp1, A = pop(A);
   B = push(B, tmp1);
    ...  }
```

If a particular nesting of function applications contains modifications of the same object in independent subexpressions, then rule 3 results in a left-to-right sequentialization of the respective modifications.

Systematically applying the transformation algorithm to the program in figure 2 results in a notation identical to our first approach as shown in figure 1.

Concerning our implementation, all additional return values introduced by the pre-processing algorithm are removed again when sequential threads of C-code are generated by the SAC-compiler. Instead, a SAC-function that modifies an object is compiled to a C-function that requires a pointer to the respective data structure. Accordingly, the application of such a SAC-function is compiled to an application of the corresponding C-function to the address of the specific data structure. In other words, a call-by-reference mechanism is applied in these cases. Thus, our short cut notation can be considered to represent a full-fledged call-by-reference parameter passing mechanism.

Nevertheless, some restrictions apply to the usage of reference parameters in a function's body. They may neither be part of the function's `return`-statement nor of any function application which might result in the consumption of the respective object. These restrictions are necessary to rule out situations that would result in the introduction of uniqueness violations by the transformation algorithm.

4 Global Objects

All objects needed by a particular function must explicitly be passed to it as arguments. This concerns objects that are directly accessed by the function itself, e.g. `stdout` by `printline` in figure 2, as well as objects which are only passed as arguments to other functions, e.g. `stdout` in `Header`. As a consequence, parameter lists are considerably extended even though most parameters are passed through without any action, especially if several function calls are nested in each other. Furthermore, it complicates the subsequent introduction of operations on objects into an existing application program, e.g. adding output statements for debugging purposes.

To overcome these deficiencies, another notational short cut is introduced into SAC: global objects which can be accessed in any function regardless of whether or not they are passed as parameters. They may be created using the new SAC instruction `objdef`, whose syntax is:

```
objdef <class> <ident> = <expr>;
```

This defines a global object of class `<class>` that is called `<ident>` and is generated by `<expr>`. Usually, `<expr>` is an application of a constructor function of `<class>`. The modified version of the case study in figure 4 illustrates the usage of global objects.

```
ClassDec File :
own:
{
funs: File open_stdout(World &);
      void fprintf(File &, string);
}


import File: all;

objdef File stdout = open_stdout(world);

void printline(string message)
{
  fprintf(stdout, message);
  fprintf(stdout, "\n");
}

void Header()
{
  printline("This is SAC !");
}

int main()
{
  Header();
  fprintf(stdout, "Hello World.\n");
  return(0);
}
```

Figure 4: Providing I/O facilities with global objects.

Since `stdout` is defined as a global object, it can be accessed by the function `printline` without being passed as a parameter. Having a closer look at the function definition of `Header`, we can observe that `stdout` has completely disappeared. It is no longer needed because `Header` itself does not send output to `stdout` and the new `printline` does not require it as an argument.

Global objects can be derived from other previously defined global objects as `stdout` is derived from `world`. In this context, `world` is considered to be the sole generic global object of class `World`. Global objects cannot be removed by

the application program. They are generated by the runtime system when program execution starts and removed when it terminates. Therefore, the function `close_stdout`, as used in the preceding versions of the case study, is no longer exported by `File`.

As pointed out above, global objects represent a notational short cut for explicitly passing all necessary objects as arguments to each function. This includes all global objects that are directly accessed by the function itself as well as those needed by functions applied in its body. We propose a transformation algorithm that eliminates all applications of the new notation in a SAC-program. When extending the pre-processing algorithm by the four additional rules outlined in figure 5, compiling phases relying on referential transparency are not affected by the introduction of global objects.

1. For each function definition except `main`:
   - Create a list of all global objects needed. These are not only the global objects accessed by the function itself but recursively all global objects needed by functions applied in its body.
   - Add all objects needed to the parameter list of the function applying the call-by-reference mechanism.
2. For each function application:
   - If the function has been modified according to the first rule:
     - Adjust the application by passing the respective objects explicitly to the function.
3. For each global object definition `objdef <class> <ident> = <expr>; :`
   - Insert the object definition `<ident> = <expr>;` right before the first original statement in the body of `main`.
   - If `<expr>` contains `world` or recursively any other global object `<object>` of class `<objclass>` that is derived from `world` :
     - Insert an application of the generic function
       `void reintegrate(<class>, <objclass> &)`
       at the end of `main`:
       `reintegrate(<ident>, <object>);`
   - Delete the global object definition.
4. If the generic global object `world` appears in any object definition:
   - Make sure that `World & world` occurs in the parameter list of `main`.

Figure 5: Transformation algorithm for eliminating global objects.

The basic idea of the transformation algorithm is given by rule 1. However, finding out which global objects are needed by a particular function is not as difficult as it may appear on first sight due to the absence of higher-order functions in SAC. Of course, all function applications must be adjusted accordingly (rule 2). Each global object definition is replaced by an object definition local to `main` (rule 3). Special attention must be paid to the correct removal of global objects related to the external environment. The generic function `reintegrate` is designed to reconstruct `world` as the environment's monolithic representation before program termination. Whenever a program is designed to interact with its environment, `world` has to be passed as a parameter to its `main`-function (rule 4).

Applying this transformation algorithm to the program of figure 4 results in a notation very similar to that of figure 2. In fact, only the application of the function `close_stdout` is replaced by an application of the generic function `reintegrate`.

Similar to the implementation of the call-by-reference mechanism, all additional parameters and return values introduced by the pre-processing algorithm are removed when generating sequential threads of C-code. Instead, a global object is compiled to an external C-variable. Thus, global objects not only represent a syntactical short cut, convenient for the application programmer, but allow for an efficient implementation as well.

However, the same restrictions apply for the usage of global objects as for that of reference parameters. Additionally, a global object may not be passed as an argument to a function that already makes use of it. As the other restrictions, this one is necessary to avoid the introduction of uniqueness violations by the transformation algorithm.

In fact, global objects may not only be defined in a program but in a module or class implementation as well. In this case, they can be exported by the respective module or class in the same way as functions. If imported into a program, such a global object may be used exactly as if it was defined by the program itself. In figure 6, this concept is illustrated by a fragment of the last version of our case study.

```
ClassDec File :
own:
{
global objects: File stdout;
funs: void fprintf(File &, string);
      void printf(string);
}

   ...
```

Figure 6: Importing global objects from classes

The global object `stdout` is declared in a new section of the class declara-

tion. The class implementation of `File` has become responsible for creating and removing `stdout` as well as for the correct application of `world`. Therefore, the constructor function `open_stdout` has become obsolete and is no longer provided by `File`. Since `stdout` is available in the implementation of `File` as well, the class may provide functions which implicitly operate on it, e.g. a function `printf` that implicitly writes to the standard output channel as in C.

5 Conclusion

In this paper we describe the mechanisms for handling states and state modifications in SAC. Due to its basis of uniqueness typing, the class concept of SAC can be safely integrated without violating referential transparency. In order to provide more comfortable means for the modification of states, two typically imperative mechanisms are adopted: call-by-reference parameters and global objects. They allow the programmer to omit some argument and return values which from a pragmatical point of view are superfluous. As a consequence, the introduction of states and state modifications into existing programs is facilitated since functions can modify states without requiring them as arguments. Besides being helpful when global objects such as counters or stacks are needed, this feature of SAC is essential whenever output operations temporarily have to be integrated for debugging purposes.

As shown in the preceding chapters, these two constructs do not violate the referential transparency since they are syntactical short cuts for purely functional expressions. However, the resulting syntax allows for another interpretation which might be more familiar to programmers who are used to imperative languages. Functions that have a call-by-reference parameter or modify global objects can be considered to perform side-effects. Furthermore, one might assume a sequential control-flow between such "'side-effecting"' functions because a particular sequence of state modifications is guaranteed although explicit data-dependencies are missing.

This dualism of a purely functional meaning on the one hand and a possible imperative interpretation on the other hand allows the programmer to specify state modifications in the style he is used to. Therefore, we hope that SAC might overcome some of the acceptance problems that functional languages normally suffer from.

Moreover, the imperative interpretation allows the compiler back-end to finally generate sequential threads of C-code that do perform these side-effects. All superfluous arguments/ return values are omitted at the implementation level, thus improving the C-code generated. As a consequence, all functions provided by the standard C-libraries can directly be used in SAC. This meets the design goal of SAC to adopt as much as possible from C and simplifies the integration of I/O into the SAC-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.

[AGP78] Arvind, K.P. Gostelow, and W. Plouffe: *The ID-Report: An asynchronous Programming Language and Computing Machine.* Technical Report 114, University of California at Irvine, 1978.

[AP95] P. Achten and R. Plasmeijer: *The ins and outs of Clean I/O.* Journal of Functional Programming, Vol. 5(1), 1995, pp. 81–110.

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

[CH93] M. Carlsson and T. Hallgren: *FUDGETS - a Graphical User Interface in a Lazy Functional Language.* In FPCA '93, Copenhagen. ACM Press, 1993, pp. 321–330.

[For94] High Performance Fortran Forum: *High Performance Fortran language specification V1.1*, 1994.

[Fra91] J. Frankel: *C* language reference manual.* Thinking Machines Corp., Cambridge MA, 1991.

[HS89] P. Hudak and R.S. Sundaresh: *On the Expressiveness of Purely Functional I/O Systems.* Technical report, Yale University, 1989.

[JW93] S.L. Peyton Jones and P. Wadler: *Imperative functional programming.* In POPL '93, New Orleans. ACM Press, 1993.

[LJ94] J. Launchbury and S. Peyton Jones: *Lazy Functional State Threads.* In Programming Languages Design and Implementation. ACM Press, 1994.

[Per91] N. Perry: *The Implementation of Practical Functional Programming Languages.* PhD thesis, Imperial College, London, 1991.

[PvE95] M.J. Plasmeijer and M. van Eckelen: *Concurrent Clean 1.0 Language Report.* University of Nijmegen, 1995.

[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.

[Sch94] S.-B. Scholz: **S***ingle* **A***ssignment* **C** – *Functional Programming Using Imperative Style.* In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia, 1994.

[Wad92] P. Wadler: *Comprehending Monads.* Mathematical Structures in Computer Science, Vol. 2(4), 1992.