


The SOF Programming Paradigm: A Sequence of Pure Functions

Antoine Bossard, Kanagawa University, Japan*

 <https://orcid.org/0000-0001-9381-9346>

ABSTRACT

Out of the four main programming paradigms, it is widely considered that functional programming is the most promising. The programming languages that implement the functional paradigm generally do so either in a pure manner, such as Haskell, or by providing a multi-paradigm programming solution, such as most Lisp dialects, in order to allow side effects, which are proscribed under the former (pure) model. Nevertheless, tracking the execution steps of such a functional program remains challenging for the programmer. In this paper, the author addresses this issue by proposing a novel programming paradigm that combines the imperative programming approach based on a sequence of instructions with the pure function approach of functional programming, the objective being to retain the advantages of both strategies. This proposal is named “sequence of functions” (SOF), and its applicability and novelty are shown hereinafter throughout various examples and experiments.

KEYWORDS

Assembly, Functional, Instruction, Paradigm, Parallel Processing

1. INTRODUCTION

One can distinguish four main programming paradigms: imperative, object-oriented, functional and logic. In this research, the focus is on imperative and especially functional programming, and more precisely between the two (see below). While imperative programming with languages such as the C and Pascal language remains mainstream, functional programming, whose importance has been acknowledged for decades (Hughes, 1989), has seen an increased interest in recent years (Bossard and Kaneko, 2019): the new versions of programming languages of the imperative and object-oriented paradigms feature mechanisms that are derived from functional programming. For example, JavaScript now provides the `Array.prototype.map` method (Terlson, 2018) and C++17 the `std::apply` function (ISO/IEC JTC 1/SC 22, 2017). As mentioned above, functional programming is at the centre of this work. With this programming paradigm, the program logic takes precedence over the control of the program execution: we say that functional programming is declarative. As a result, tracking the state of the program and tracing its execution is challenging for the programmer. And especially with functional languages that feature lazy evaluation.

DOI: 10.4018/IJSI.309965

*Corresponding Author

This article published as an Open Access article distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/4.0/>) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

In this paper, we take on this issue by proposing a programming paradigm – and an implementing programming language – that features “instructions” executed in a sequence in an imperative programming style manner (thus comparable to a program in the assembly language), but, importantly, with these instructions being pure functions in order to retain the referential transparency advantages of pure functional programming. This proposal is named “sequence of functions” (SOF).

In addition, thanks to this approach based on a sequence of instructions, we project to facilitate the parallel execution of functional programs, for example when data are to be disseminated across the interconnection network of a massively parallel system such as the torus network binding RIKEN’s Supercomputer Fugaku (Bossard and Kaneko, 2020). This issue is indeed notorious (Ahn and Han, 2000).

The proposed programming paradigm and language stay purposefully simple, in a way that may remind the reader of a Reduced Instruction Set Computer (RISC) approach (Patterson and Ditzel, 1980), a technology which is used for example by the aforementioned Supercomputer Fugaku – it is equipped with ARM computing nodes (RIKEN, 2020) – and which may be deemed the future given its ecological friendliness (green computing) (Aroca and Gonçalves, 2012).

SOF designates at the same time a programming paradigm and a sample implementation of this paradigm, that is a programming language. The SOF approach positions itself between the imperative and functional paradigms, as shown in Figure 1. Note that this paper is an extended version of Bossard (2020): it significantly clarifies the SOF proposal, for instance by situating this new paradigm in the context of the existing ones, concretely describes the realisation of a SOF compiler and conducts thorough experimental evaluation whose results are subsequently discussed. These are the main, and major, differences with Bossard (2020).

The rest of this paper is organised as follows. Related works and preliminaries are reviewed in Section 2. The proposed paradigm and language are detailed in Section 3. Additional examples are given in Section 4. A compiler for the proposed paradigm is described in Section 5. Evaluation is conducted in Section 6 and the paper is concluded in Section 7.

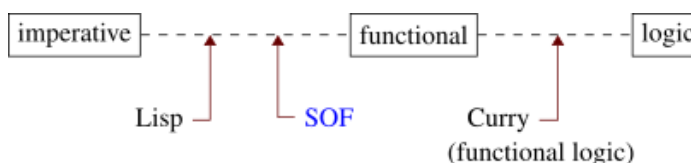
2. RELATED WORKS AND PRELIMINARIES

We review in this section the conventional approaches regarding operation sequencing in functional programming. It is recalled that this is a feature inherent in imperative programming. A large part of functional languages are multi-paradigms; this is the case of Lisp and its numerous dialects such as Scheme and Racket. These languages allow side effects, and thus sequences of operations, for example with the `begin` function (Sperber et al., 2009).

The other functional languages are pure, such as the Haskell language, with thus side effects being proscribed. For instance, Haskell relies on monads and `do` expressions to achieve somehow comparable sequences of operations (Marlow, 2010).

The proposed SOF approach further extends this idea; a more detailed comparison with the conventional approaches will be conducted in Section 6.1. In addition, our approach is arguably simpler than Gifford and Lucassen’s (1986) which is based on the “fluent” languages which combine different sub-languages for the imperative and functional paradigms, and of which we found trace in some Lisp dialects such as Common Lisp (Burgemeister, 2018) and Scheme (Sperber et al., 2009).

Figure 1. Position of the SOF approach with respect to other programming paradigms and related languages



Relevant definitions and notations are detailed next.

Definition 1: A side effect in programming is the modification of a value other than the return value.

Definition 2: A pure function in programming is a function that has no side effect.

A pure function is also called a mathematical function.

The situation of the conventional approaches is as follows. On the one hand, the program execution state is key to imperative programming, side effects thus being ubiquitous in imperative languages such as assembly and C: a variable assignment statement such as $x = 0$; is an example of side effect in C.

On the other hand, the functional languages such as Haskell that stay pure, that is that do not compromise when implementing the functional paradigm, involving exclusively pure functions with thus no side effect allowed, rely on monads to realise I/O operations without side effects.

Nonetheless, it is a challenge for the programmer to deal with such a complete absence of side effects, which is why compromising, that is multi-paradigm functional languages such as Lisp and its major dialects (e.g. Scheme, Racket), have been continuously popular (IEEE Spectrum, 2019). Effectively, they provide the developer with the possibility (whose usage ought to stay exceptional) to sequence operations in an imperative manner in the midst of the otherwise functional program. This is done for example for I/O operations (e.g. Scheme's `write-char` port procedure) and with the `begin` function previously mentioned.

3. THE SOF PROGRAMMING PARADIGM AND LANGUAGE

We describe in this section the proposed programming paradigm and language, SOF, whose approach is to rely on a sequence of instructions when writing the program, thus in an imperative style, with however instructions being pure functions, thus in a functional style.

3.1 Basic Constructs

First and foremost, to each instruction of a SOF program is assigned a label, which is nothing more than a name given to the instruction. Such a name is unique, that is, the same name cannot be given to more than one instruction. But an instruction can be given several names, which thus become synonyms. For convenience, a default label is assigned to each instruction (e.g. with simple numbering), which is thus comparable to the instruction address as found for instance in assembly. This labelling may also remind of first generation BASIC programming languages. An instruction consists of one or several nested statements.

By principle, each instruction, which is a pure function as just recalled, takes a stack as first parameter (i.e. the input stack) and its return value is also a stack (i.e. the output stack). Since dealing with pure functions, with thus no side effects, such stacks are immutable objects: each instruction thus returns a new stack each time. The output stack is implicitly named `stack` as it is necessarily used by the next program instruction.

The program instructions are conventionally executed in a top-to-bottom order. The final result of such functional program is at the top of the stack that is returned by the last instruction. As found in other languages, the state of the final stack can be used to infer an unsuccessful program execution: for instance, when the last stack contains no element or at least two.

Therefore, a statement of a SOF program has by principle the following form:

```
function(stack, par1, par2, ...)
```

with `function` the name of the function to be evaluated, of first parameter the input stack and of return value the output stack in accordance with the paradigm rules. And, in addition to the input

stack there may be additional (optional) parameters. Parameter passing is further discussed later in this section.

For convenience, arithmetic functions are available in various forms, that is with more or less parameters. The case of the `addition` function is shown below:

- The statement `addition(stack, 3, 2)` is to push the result (5) on the stack without popping.
- The statement `addition(stack, 3)` is to read and pop the topmost stack element and to add it to the second parameter. Finally, the sum (result) is pushed on the stack.
- The statement `addition(stack)` is to successively read and pop twice the topmost stack element and to sum their respective value. Finally, the sum (result) is pushed on the stack.

One should note that these variants need not be restricted to binary operations. Such usage of the stack may remind the reader of the zero address instruction format which is typical of stack machines (Koopman, 1989).

Next, we address the conditional statement. As per the previously stated SOF guidelines, an `if` statement can be defined as follows:

```
if(stack, condition, then_statement, else_statement)
```

And again as per the SOF paradigm's functional nature, the "then" and "else" statements are both required (which is not the case with the imperative paradigm). The conditional statement is thus slightly particular in that its return value is the result (i.e. the output stack) of either the "then" statement or the "else" statement whereas the return value of other, regular statements is implicit. Nonetheless, the return value of the conditional statement is also implicitly named `stack`.

Relying on bindings induced by pattern matching (a.k.a. pattern bindings) has proven convenient in some situations. This is the case for example when expressing the condition of an `if` statement as detailed previously. The scope of bindings created with pattern matching is the whole instruction, that is, the place where the binding is declared inside the instruction is irrelevant. Such pattern bindings are declared as follows:

```
if((e:_)#stack, e == 0, ..., ...)
```

where the parenthesised expression that is on the left of the sharp symbol (`#`) is the declaration of the bindings that are to be realised by pattern matching on the object, here `stack`, that appears on the right of the sharp symbol. The syntax of the pattern matching expression is comparable to Haskell's for lists: the object that is on the left of the colon (`:`) is the topmost stack element and the object that is on the right of the colon represents the other elements (i.e. the rest of the stack). Besides, the underscore symbol (`_`) is a wildcard: it matches anything but creates no binding. Hence, in the above pattern expression, one single binding is created, named `e`, and it represents the topmost element of the stack `stack`. This new binding is used to express the condition (`e == 0`) of this `if` statement. Finally, it can be noted that since relying on pure functions, with thus no side effect (immutable objects), the declaration and usage of pattern bindings as previously described remain sound.

Next, we recall that one primary objective of the SOF approach is to render the program execution easier to track. To this end, the SOF paradigm is based on a top to bottom programming style (unlike, for instance, declarative languages such Haskell and Prolog). Furthermore, a unique label is assigned to each instruction. Hence, it is meaningful to enable branching with a `goto` statement. The `goto` statement takes as parameter a stack as usual and a label that is used as branching address. The `goto` statement is thus particular in that it does not have its own return value. Instead, it indirectly returns the value of the instruction that corresponds to its branching destination. The stack set as parameter

to the `goto` statement is to define the stack that is to be used by the subsequent instructions. The `goto` statement is exemplified below:

```
1: push(stack, 1)
2: goto(stack, [4])
3: addition(stack, 1) // skipped instruction
4: subtraction(stack, 1)
```

The result of this program is thus 0.

It can be noted that the stack given as parameter to the `goto` statement could be omitted in some cases as the stack binding necessarily has already been made in the preceding instruction – this is the case in the above example. Nonetheless, it is often convenient to explicitly give a stack to the `goto` statement as illustrated in the following sections.

Function calls complete this presentation of the SOF language: their parameter is a stack and so is their return value. In the case of additional parameters, they are simply pushed atop the stack beforehand. Therefore, the sole difference between a function call and a `goto` statement is that the last instruction of a function is a `ret` statement: it used to set the stack that is the return value and to return to the calling statement. Examples are given below.

It can be noticed that a “native” function (i.e. that is not defined by the SOF programmer) may take parameters in addition to the input stack. At the exception of a `goto` statement, a function call can indeed be written equivalently as `function(stack, par1, par2, ...)`: this notation simply means that the parameters `par1, par2...` are first pushed onto the stack before a call to `function(stack)`. For clarity, a call to a user-defined function will hereinafter be preceded by explicit push operations for its parameters, if any, other than the input stack.

Finally, it can be noted that several statements can be combined (nested) into one, like `push(push(...))` (i.e. which induces one instruction) instead of two successive statements `1: push(), 2: push()` (i.e. which induces two instructions), but this weakens the benefit of the `goto` statement, and more generally of the SOF paradigm (i.e. facilitating tracking the execution of the program), since an instruction that consists of such a combining statement is “addressable” as one single instruction whereas successive statements induce distinct instructions which can thus each be addressed separately.

3.2 Repetition

In this section, automated processing by repetition is illustrated with a pedagogical example: an initial value is repetitively incremented, ten times in total. Three different programming methods for this same sample calculation are described. The first program is named R1; it is based on a `goto` statement:

```
1: push(stack, 0) // counter initially at 0
2: push(stack, 5) // result initially at 5
3: addition(stack, 1) // pop, add 1, push
4: flip(stack) // flip the two topmost elements
5: addition(stack, 1) // increment counter
6: if((e:_)#stack, e == 10, pop(stack), goto(flip(stack), [3]))
```

The result of the whole program is the topmost element of the stack that is returned last, here 15.

The two increment operations (counter, result) can also be directly included in the conditional statement. This second version of the program, still relying on the `goto` statement, is named R2:

```
1: push(stack, 5) // result initially at 5
2: push(stack, 0) // counter initially at 0
```

```
3: if((e1:e2:es)#stack, e1 == 10, pop(stack),  
goto(addition(addition(es, e2, 1), e1, 1), [3]))
```

Finally, this calculation can be conducted with a function call in place of a `goto` statement. This third program is named R3:

```
f:  
1: if((e1:e2:es)#stack, e1 == 10, ret(pop(stack)),  
ret(f(addition(addition(es, e2, 1), e1, 1)))) // e1: the counter;  
e2: the value to increment  
start:  
2: push(stack, 5) // result initially at 5  
3: push(stack, 0) // counter initially at 0  
4: f(stack) // function call
```

To the instruction labelled 1 is assigned a new, synonym label `f` used for readability: the functional call could indeed similarly be written `[1] (stack)` reminding of C's function pointers. The other synonym label, `start` declares the first instruction to be executed (i.e. the entry point).

3.3 On the `goto` and `return` Statements

A `goto` statement is valid only as a top-level statement (i.e. not as a parameter of another statement) or as the “then” or “else” statement of an `if` statement that is itself a top-level statement (i.e. the `if` statement is not a parameter of another statement). That is to say that a `goto` statement is not allowed inside a statement other than a top-level `if` statement; effectively, a `goto` statement as parameter is meaningless since the calling statement would not be evaluable (and only partly evaluable if the `goto` statement is inside an `if` statement). So, there are two allowed usages for the `goto` statement: `goto` as a top-level statement, or `goto` inside an `if` that is a top-level statement. Besides, since a `goto` statement cannot be inside a statement other than a top-level `if`, a `goto` statement inside such an `if` is only allowed as the “then” or “else” statement, and not *inside* the “then” or “else” statement. For example, `if(..., goto, ...)` is fine but `if(..., push(goto, ...), ...)` is not.

This discussion on the `goto` statement is essential regarding the SOF paradigm, and has important implications for instance when building a SOF compiler as detailed in Section 5.

For the same reasons, the same restrictions apply to the `ret` (return) statement, which is for user-defined functions. A label which is an atom, that is like `f` :, declares a function; it is followed by a newline character `\n` and the instructions of the function. The end of the function is thus detected by a top-level `ret` statement or a top-level `if` statement that has a `ret` statement as its “then” and “else” statements. Functions are declared in the preamble part of the SOF program, that is, at the top of the program, before the entry point, which is marked with the label `start` : on its own line.

Also note that unlike SOF native functions, whose parameters are directly passed to the function (i.e. not in the SOF stack), user-defined functions' parameters are passed on the stack as explained. And as in C, a `goto` statement cannot take a label outside of its “scope”: if inside a user-defined function, its jump label needs to be also inside that function, and if not inside a user-defined function, that is inside the main program, its jump label cannot point at an instruction that is inside a user-defined function.

4. REAL-WORLD EXAMPLE

In this section, we consider the classic functional program example: the factorial function. We give three possible implementations on a par with Section 3.2. The first program, named F1, is based on the `goto` statement:

```
1: push(stack, 1) // result initially 1
2: push(stack, 5) // calculation of 5!
3: if((e1:e2:es)#stack, e1 <= 1, pop(stack), goto(subtraction(mul
multiplication(es, e1, e2), e1, 1), [3]))
```

The initialization of the intermediate result is comparable to tail recursion as used conventionally in functional programming.

The next program, named F2 and still relying on the `goto` statement, this time realises the arithmetic operations outside of the conditional statement:

```
1: push(stack, 5) // calculation of 5!
2: push(stack, 1) // result initially 1
3: multiplication((_e2:_)#stack, e2)
4: flip(stack) // flip the two topmost elements
5: subtraction(stack, 1) // counter update
6: if((e1:_)#stack, e1 <= 1, pop(stack), goto(flip(stack), [3]))
```

The third and last program, named F3, this time relies on a function call to a recursive function:

```
f:
1: if((e1:e2:es)#stack, e1 <= 1, ret(pop(stack)), ret(f(subtrac
tion(multiplication(es, e1, e2), e1, 1)))) // e1: counter; e2:
intermediate result
start:
2: push(stack, 1) // result initially 1
3: push(stack, 5) // calculation of 5!
4: f(stack) // function call
```

5. SOF COMPILER AND SDK

We have implemented a compiler for the SOF programming paradigm and induced programming language. This compiler generates standard C code, which can thus subsequently be compiled to machine code by a C compiler such as the one bundled with the GNU Compiler Collection (GCC). The most interesting part of the SOF compiler is arguably the handling of `goto` statements and the bindings induced by pattern matching. Native functions such as `push`, `pop` and `flip` are directly implemented in C for obvious performance reasons.

This SOF compiler is rather conventionally produced as described in Figure 2: scanning for tokens is done by flex and parsing of the SOF grammar is realised by bison, two Unix text processing tools (Levine, 2009). The respective outputs of flex and bison are, together with additional helper C functions, set as input of the C compiler, which eventually produces the machine code of the SOF compiler.

The flow of the generation of machine code that corresponds to a SOF program is shown in Figure 3: the source code of the SOF program is passed to the SOF compiler which generates intermediate C code. This C code is in turn passed together with C functions provided by the SOF software development kit (SDK) to the C compiler, which produces the machine code that corresponds to the initial SOF program.

6. EVALUATIONS

In this section, the contribution and validity of the SOF proposal are first shown, before measuring and comparing its performance.

Figure 2. The generation process of the SOF compiler

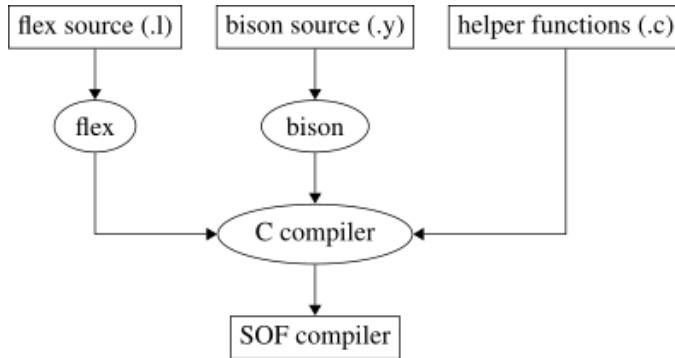
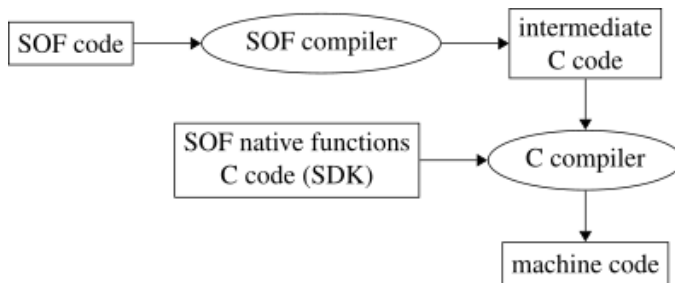


Figure 3. The flow of the generation of machine code that corresponds to a SOF program



6.1 SOF Versus Related Paradigms and Languages

First, we compare the SOF paradigm and language to related paradigms and languages. To this end, we consider the real-world example of Section 4: the implementation of the factorial function.

In assembly, this function can be implemented as follows (x86 assembly for FASM):

```

mov cx, 5 ; counter set for 5!
mov ax, 1 ; result initially 1
@@: ; anonymous label
mul dx ; ax <- ax * dx
sub cx, 1 ; counter update
cmp cx, 1 ; counter check
jg @b ; jump if counter > 1
; the final result is in ax
  
```

It can be noted that the numerous side effects, such as the modification of a register or flag, render the program logic difficult to understand.

Then, we consider a Haskell implementation of the factorial function:

```

f:: Integer -> Integer
f x = if x <= 1 then 1 else res
  where
    rec = f (x - 1)
    res = x * rec
  
```


Unlike the above assembly implementation, the execution of the program, that is the sequence (order) in which the operations are evaluated, is arguably difficult to follow. Furthermore, one should note that this Haskell program only defines the factorial function: this function needs to be called from within another Haskell expression or at the Haskell REPL prompt. In fact, function calls are not allowed to be at the top of the source tree in Haskell. This further harms the understanding of the sequence of operations.

A Scheme implementation for the factorial function is finally given, featuring, unlike Haskell, basic sequencing:

```
(define (f x) ; function definition
  (if (<= x 1) 1
      (* x (f (- x 1)))))
(f 5) ; function call
```

This Scheme implementation is indeed between the assembly one and the Haskell one: the definition of the function `f` and the function call are executed sequentially, from top to bottom (in fact, a call of a function before its definition produces an error). The proposed SOF approach goes further by enabling `goto` statements to alter the execution sequence by branching it to an instruction of the program as described in Section 3. We have thus shown with a real-world example the contribution and validity of the SOF proposal.

6.2 Experimental Evaluation

We have evaluated our proposal by conducting several experiments that measure the program execution time and the program memory size when using the SOF paradigm and language compared to other, conventional approaches. These two criteria are conventional measurements when dealing with programming paradigms and languages. To this end, we have reused the previously presented SOF sample programs:

- The three programs R1, R2 and R3 that repetitively increase a counter;
- The three programs F1, F2 and F3 that implement the factorial function.

A program corresponding to the calculation conducted by the R1, R2, R3 programs has been for comparison purposes implemented in the functional languages Racket (multi-paradigm) and Haskell (purely functional), and in C directly. Two C implementations were considered: one iterative, thus closer to the R1 and R2 programs, and the other recursive, thus closer to the R3 program. Similarly, a program corresponding to the factorial calculation (i.e. the F1, F2, F3 programs) has been for comparison purposes implemented in Racket and Haskell, and in C directly. Two C implementations were considered: one iterative, thus closer to the F1 and F2 programs, and the other recursive, thus closer to the F3 program. Note that the compiled Haskell code has a `print` statement as main expression to return an `IO()` object as required.

The Racket compiler used in these experiments was `raco make`, the compiler that is bundled with the Racket development environment, and the Haskell compiler was the Glasgow Haskell Compiler (GHC). The C compiler used in these experiments was that of the GCC. It was thus used to obtain machine code from both the SOF programs' intermediate C code (refer to Section 5) and from the C implementations of the programs used in these experiments.

6.2.1 Experiment 1: Program Execution Time

The first experiment consisted in the measurement of the execution time of the considered programs. The execution time of the Racket and Haskell implementations has been measured when the program

is interpreted (i.e. not compiled in advance to machine code) and when a compiled copy is run (i.e. when executing the machine code output by the Racket and Haskell compilers).

Time measurements were realised in the following experimental conditions: each program was automatically run 1,000 times, invoked repetitively by a shell script except the interpreted Racket and Haskell programs that were run repetitively 1,000 times with an additional, specially defined Racket and Haskell function. The programs were run in a Debian GNU/Linux 10 (buster) virtual machine on a Windows 10 Home (64-bit) computer equipped with an Intel Core i7-6700 CPU clocked at 3.4 GHz and 16 GB RAM, except for the Racket and Haskell programs that were run on the same computer but natively (i.e. not inside the Debian virtual machine). In the Debian virtual machine, the execution time was retrieved by the Bash `time` standard utility; the real time value was retained for homogeneity across all the measurements. The execution time of the interpreted Racket programs was retrieved by the standard Racket function `time`, and that of the interpreted Haskell programs by the information provided by the GHCi interpreter when activating the: `set +s` setting. Finally, the execution time of the compiled Racket and Haskell programs was recorded with the PowerShell `cmdlet Measure-Command`.

To obtain an execution time value that is as stable as possible, each 1,000 times run was done 3 times. The full results are given in the Appendix (Table 3) and the averaged results (i.e. the average execution time of 1,000 program runs) are shown in Table 1. In this table, the column label “i-Racket” (resp. “c-Racket”) stands for interpreted (resp. compiled) Racket and “i-Haskell” (resp. “c-Haskell”) stands for interpreted (resp. compiled) Haskell. It is recalled that we considered two C implementations, one iterative the other recursive: the former applies to R1, R2 and F1, F2, and the latter to R3 and F3, hence the values of the last column of this table.

6.2.2 Experiment 2: Program Memory Size

The second experiment aimed at measuring the memory size taken by the machine code generated for the considered programs (obviously, the interpreted versions of the Racket and Haskell programs are not considered this time). The Racket programs were compiled as stand-alone programs and the Haskell and C programs were compiled with the default linker settings. It can be noted that the machine code generated by the Racket compiler was split into four files: the executable file and three library files; the aggregate memory size was considered. The memory size of the output machine code is detailed in Table 2.

6.2.3 Results Discussion

Regarding the first experiment (program execution time), it can be noticed that the execution time of the machine code induced by the SOF programs is on par with fastest conventional solutions, such as the machine code obtained from the C implementations.

Table 1. The average execution times of 1,000 program runs (in milliseconds)

Program	SOF	i-Racket	c-Racket	i-Haskell	c-Haskell	C
R1	553	52	318,285	470	16,598	519
R2	543					
R3	562					537
F1	538	52	322,090	533	16,848	534
F2	533					
F3	537					518

Table 2. The memory size taken by the machine code of programs (in bytes)

Program	SOF	c-Racket	c-Haskell	C
R1	17,136	13,160,767	14,277,849	16,464
R2	17,136			16,496
R3	21,256			
F1	17,136	13,160,767	14,277,197	16,464
F2	21,232			16,488
F3	21,256			

On the one hand, interpreted Racket and Haskell programs should be executed much faster since there is no process switching (i.e. triggered by the running of an executable file) at all. The fact the interpreted Racket and Haskell programs are not executed much faster shows that our machine code is efficient. On the other hand, interpreted code needs to be evaluated which could slow the execution down. But since the tested programs are very short, it is reasonable to assume that process switching takes more time than code evaluation. Hence, the fact that the interpreted code is not much faster than the machine code induced by SOF programs is a good indicator of the performance of the proposed SOF approach and compiler. On a side note, the times measured for the interpreted Racket code are very low. The fact that they are about ten times lower than those obtained with the corresponding interpreted Haskell programs indicates that they should not be considered in the same manner as the other recorded times. It is highly probable that the interpreted code is implicitly compiled before execution, which would explain the difference with equivalent Haskell code and execution method. Even more so as Haskell compiled code is usually significantly faster than Racket compiled code, as shown for instance in Table 1. Consequently, it is sound to say that, although obviously not as feature-complete, the machine code of SOF programs is thus extremely fast compared to Haskell and Racket machine code.

Regarding the second experiment (program memory size), considering the machine code induced by the SOF programs R1 and R2, a slight increase (4%) in size can be noticed compared to the machine code generated from the plain C program. When relying on a recursive function (i.e. the SOF program R3), the increase is larger: 29% when compared to the corresponding plain C program. When considering the SOF programs F1 and F2, the increase is 4% in the case of F1 and 29% in the case of F2 when compared to the corresponding plain C program. This significant difference can be explained as follows: the SOF program F2 has twice the number of instructions that the SOF program F1 has. And again, when relying on a recursive function (i.e. the SOF program F3), the increase is 29% when compared to the corresponding plain C program.

Regarding the machine code generated from the Racket and Haskell programs, as mentioned, the Racket programs were compiled as stand-alone programs (just as the SOF programs), which explains the significantly larger size of the machine code: some Racket libraries are statically linked with the machine code of the program (executable file) and three dynamically linked libraries were added to make the program system-independent. Although not specifically mentioned at compile time, it is reasonable to assume the same for the Haskell compiler given the large machine code size.

As a result, it is sound to conclude that the machine code produced from the SOF programs is near optimal, although the program writing style can impact the machine code size, especially when considering the huge machine code obtained from the Racket and Haskell compilers.

7. CONCLUSION

Although pure functions as in (pure) functional programming have very interesting properties, for instance helping to achieve referential transparency, it is notoriously difficult to track the execution of such a program. In this paper, aiming at addressing this issue, we have proposed the SOF programming paradigm and language. Not only is SOF a novel approach, it is also promising in that, by design, it features both the advantages of a program based on pure functions, and those of imperative programming with an easy to follow execution sequence. The contribution and validity of the proposal has been shown throughout various examples and by comparing it to conventional, related approaches. Besides, the performance of SOF has been empirically measured and discussed: it is on par with the fastest conventional approaches. Regarding future works, it would be interesting to further measure the performance of SOF, for instance when applying it to larger scenarios.

FUNDING

This research was partly supported by a Grant-in-Aid for Scientific Research (C) of the Japan Society for the Promotion of Science under grant no. 19K11887.

CONFLICTS OF INTEREST

The author declares no conflict of interest.

REFERENCES

- Ahn, J., & Han, T. (2000). An analytical method for parallelization of recursive functions. *Parallel Processing Letters*, 10(4), 359–370. doi:10.1142/S0129626400000330
- Aroca, R. V., & Gonçalves, L. M. G. (2012). Towards green data centers: A comparison of x86 and ARM architectures power efficiency. *Journal of Parallel and Distributed Computing*, 72(12), 1770–1780. doi:10.1016/j.jpdc.2012.08.005
- Bossard, A. (2020). Programming with a sequence of pure functions. *Proceedings of the Sixth International Conference on Electronics and Software Science (ICESS; 11, 17, 18 December)*, (pp. 1–6).
- Bossard, A., & Kaneko, K. (2019). A new methodology for a functional and logic programming course: On smoothening the transition between the two paradigms. In *Proceedings of the 20th Annual SIG Conference on Information Technology Education (SIGITE; Tacoma, WA, USA, 2–5 October)* (pp. 63–68). Association for Computing Machinery. doi:10.1145/3349266.3351408
- Bossard, A., & Kaneko, K. (2020). Cluster-fault tolerant routing in a torus. *Sensors*, 20(11), 3286. doi:10.3390/s20113286 PMID:32526955
- Burgemeister, B. (2018, October). *Common Lisp Quick Reference*. Revision 148.
- Gifford, D. K., & Lucassen, J. M. (1986). Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP; Cambridge, MA, USA, 4–6 August)* (pp. 28–38). Association for Computing Machinery.
- Hughes, J. (1989). Why functional programming matters. *The Computer Journal*, 32(2), 98–107. doi:10.1093/comjnl/32.2.98
- IEEE Spectrum. (2019). *Interactive: The top programming languages*. <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019>
- ISO/IEC JTC 1/SC 22. (2017, December). *Programming languages – C++*. Technical Report ISO/IEC 14882:2017, 5th edition, International Organization for Standardization.
- Koopman, P. J. Jr. (1989). *Stack Computers: the new wave*. Ellis Horwood.
- Levine, J. R. (2009). *flex & bison*. O'Reilly Media.
- Marlow, S. (2010, April). *Haskell 2010 Language Report*. Academic Press.
- Patterson, D. A., & Ditzel, D. R. (1980, October). The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6), 25–33. doi:10.1145/641914.641917
- RIKEN. (2020). *Japan's Fugaku gains title as world's fastest supercomputer*. https://www.riken.jp/en/news_pubs/news/2020/20200623_1/
- Sperber, M., Dybvig, R. K., Flatt, M., van Straaten, A., Findler, R., & Matthews, J. (2009). Revised⁶ report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1), 1–301. doi:10.1017/S0956796809990074
- Terlson, B. (2018, June). *ECMAScript 2018 language specification*. Technical Report ECMA-262, 9th edition, Ecma International.

APPENDIX

Experiment One's Detailed Results

The measured execution times are detailed in Table 3.

Table 3. Detail of the execution times measured for the 1,000 runs for each program (in milliseconds)

Program	Run #	Runs	SOF	Racket (inter- preted)	Racket (compiled)	Haskell (inter-preted)	Haskell (compiled)	C
R1	1	1,000	563	run #1: 47 run #2: 62 run #3: 47	run #1: 320,576 run #2: 317,158 run #3: 317,120	run #1: 470 run #2: 470 run #3: 470	run #1: 17,521 run #2: 16,044 run #3: 16,229	run #1: 523 run #2: 512 run #3: 522
R1	2	1,000	563					
R1	3	1,000	534					
R2	1	1,000	518					
R2	2	1,000	522					
R2	3	1,000	590					
R3	1	1,000	626					516
R3	2	1,000	529					535
R3	3	1,000	530					561
F1	1	1,000	531	run #1: 47 run #2: 62 run #3: 47	run #1: 328,630 run #2: 318,267 run #3: 319,373	run #1: 520 run #2: 550 run #3: 530	run #1: 18,174 run #2: 16,143 run #3: 16,226	run #1: 550 run #2: 532 run #3: 520
F1	2	1,000	327					
F1	3	1,000	556					
F2	1	1,000	544					
F2	2	1,000	527					
F2	3	1,000	528					
F3	1	1,000	538					517
F3	2	1,000	537					518
F3	3	1,000	535					519