Design Principles of an Embedded Language (EFL) Enabling Well Defined Order-Independent Execution

Moshe Goldstein
Flexible Computation Research Lab
Jerusalem College of Technology
Havaad Haleumi 21
Jerusalem 9372115, Israel
goldmosh@g.jct.ac.il

David Dayan
Flexible Computation Research Lab
Jerusalem College of Technology
Havaad Haleumi 21
Jerusalem 9372115, Israel
dayandav@g.jct.ac.il

Max Rabin
Flexible Computation Research Lab
Jerusalem College of Technology
Havaad Haleumi 21
Jerusalem 9372115, Israel
rabin.max@gmail.com

Devora Berlowitz
Flexible Computation Research Lab
Jerusalem College of Technology
Havaad Haleumi 21
Jerusalem 9372115, Israel
devora32@gmail.com

Or Berlowitz
Flexible Computation Research Lab
Jerusalem College of Technology
Havaad Haleumi 21
Jerusalem 9372115, Israel
berlo.berlo@gmail.com

Raphael B. Yehezkael
Flexible Computation Research Lab
Jerusalem College of Technology
Havaad Haleumi 21
Jerusalem 9372115, Israel
rafi@g.jct.ac.il

ABSTRACT

Parallel programming platforms are heterogeneous and incompatible; a *common* approach is needed to free programmers from platforms' technical intricacies, allowing flexible execution in which sequential and parallel executions produce *identical* results. The execution and programming model of an embedded flexible language (EFL), which implement this common approach, are presented. EFL allows embedding of deterministic parallel code blocks into a sequential program, written in *any* host language. EFL programming model constructs are presented. An EFL implementation of the Reduce Parallel Design Pattern is presented. With EFL we aim to implement safe and efficient parallel execution, in software, hardware, or both. Consequences of Rice's theorem regarding parallel computation are discussed. These consequences severely restrict what can be checked at compile time. An approach is proposed for circumventing these restrictions.

CCS CONCEPTS

• **Computing Methodologies → Parallel Computing Methodologies → Parallel Programming Languages**

KEYWORDS

Parallel programming, parallel design pattern, flexible computation.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ECBS '17, August 31-September 1, 2017, Larnaca, Cyprus
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4843-0/17/08…$15.00
https://doi.org/10.1145/3123779.3123789

M. Goldstein, D. Dayan, M. Rabin, D. Berlowitz, O. Berlowitz, R. B. Yehezkael. 2017. In *Proceedings of the 5th European Conference on the Engineering of Computer Based Systems, Larnaca, Cyprus, Aug 31st-Sep 1st 2017 (ECBS 2017)*, 8 pages.

DOI: 10.1145/123 4

1 INTRODUCTION

We are at the beginning of a paradigm shift [1] in programming, from serial to parallel, and there is a need for programming languages providing the programmer with appropriate programming abstractions. Programming languages designed for writing parallel or concurrent programs can be classified by the demands they make on the programmer. At one extreme, pure functional languages allow a programmer to disregard concurrency coordination, enabling well-defined parallel and unordered sequential execution, freeing the programmer from considering all possible execution orders. At the opposite extreme, sequential imperative languages force the programmer to consider in detail many possible execution orders. In between these extremes, languages can be designed where a compiler, debugger and run-time environment reduce the burden on the programmer.

The Embedded Flexible Language (EFL) which is presented here supports well-defined *order-independent* execution, in the spirit of Flexible Algorithms [2]. EFL's approach to parallel programming is as follows (see the code example below): in each program unit, (a) the *sequential* parts of the code (including all the Input/Output) are written in the host language, and (b) the *parallel* parts of the code are written inside embedded language blocks. EFL order-independent execution is enabled by scope rules enforced inside the EFL blocks.

A pre-compiler is needed to translate EFL embedded blocks of code into native parallel code in the host language. In our earlier paper [3] the implementation and testing of such a pre-compiler was discussed.

The idea behind EFL is to create an embedded language that looks and feels like a sequential language, but in fact gets executed in parallel by the host language run-time environment. This provides an abstraction layer over the actual parallel platform (Python's Multiprocessing module, MPI, openMP, Cilk, etc) upon which the program will run. EFL provides the average programmer with the tools to create parallelized code to optimize the use of multiple CPUs, in a simple and safe way that does not require the programmer to learn about the complications and intricacies of parallel platforms' programming.

When deciding about the design and implementation of EFL, the following points were identified and taken into account:

a) Parallel algorithms should be only slightly harder to write than sequential algorithms, reducing the parallel programming burden - using implicit rather than explicit parallelism.
b) Parallel programming language implementation should be simplified, enabling a small language to be embedded in *any* sequential imperative language which has parallel programming capabilities.
c) Because of the above two considerations, we decided to design a notation which is not too far from sequential imperative programming syntax and style, but encourages programmers to write code which enables well defined parallel execution.
d) Excess of parallelism and its associated inefficiencies should be avoided.

Thus, the purpose of EFL has been to provide programmers with a *universal* programming tool that does not require extensive training; so, they could utilize their familiarity with C-like languages when learning EFL. Through familiar constructs such as *if* conditions, *for* loops, and function calls, a programmer with experience in sequential imperative languages should have no problem getting started in EFL. In addition to the iterative constructs familiar to the programmer, some additions were made to allow implementation of advanced parallel algorithms. There are some limitations in EFL that will be new to programmers. Those limitations are imposed on the language due to the requirements of the EFL programming model principles and the order-independent execution model (see section 2 below).

The remainder of the paper is organized as follows. Section 2 covers the EFL execution and programming models. Section 3 describes the EFL grammar and semantics. Section 4 deals with parallel programming patterns in EFL. Section 5 deals with consequences of Rice's theorem regarding parallel computation. Section 6 covers related work, Section 7 provides conclusions, and Section 8 suggests further work.

2 EFL's EXECUTION AND PROGRAMMING MODELS

EFL's *execution model* is based on the concept of *Flexible Computation* [2]. A key aspect of *flexible computation* is its well-defined order-independent flexible execution, i.e. the values read during program execution, are independent of the permitted program execution orders. The values read during parallel and/or sequential program execution of a program complying with certain properties, should be identical. Flexible execution is enabled by the EFL programming model which enhances the scope rules of variables by defining where they may be initialized, where they may be updated, where they may be read. The EFL *programming model* is an abstraction of the EFL execution model whose foundations are essentially based on three kinds of limitation rules enforced in order to ensure *deterministic* parallelization:

(a) Only "*pure*" function calls may be used by the programmer. A *"pure"* function has no side-effects. That means no global variables, no Input/Output, etc. "*Pure*" functions only perform calculations, thus, always produce and return a value. Without a return value, a "*pure*" function does not provide any utility to the program [4]. Therefore, functions called from an EFL block cannot be a void function. This limitation was imposed in order to enable a well-defined parallelism. By allowing only "*pure*" functions, there is no chance for different parallel threads of execution to cause conflict. It is the responsibility of the programmer to use "*pure*" functions only. The currently implemented EFL pre-compiler does not check this.

(b) *In* and *Out* variables (but not *InOut* variables!). Let's define a variable whose value is read within an EFL block as *In*, meaning that it is an input to the block; similarly, a variable to which a value is assigned from within an EFL block is designated as *Out*, meaning that the value is an output of the EFL block. An *In* variable cannot be written to and an *Out* variable cannot be read. This limitation prevents problems of race conditions and nondeterministic execution [6]. If the ability to read and write to variables was unrestricted, a situation could arise where at one point in the code the programmer assigns a value to a variable,

```
x = f(a);
```

and then reads that variable in another subsequent line,

```
y = f(x);
```

This would be legal in languages whose execution model is sequential. In a programming language with an unordered or parallel execution model, this would be problematic because the order of execution is unknown. To ensure that the execution of EFL code is deterministic, this is not allowed. Therefore, if a variable is assigned to anywhere in an EFL-block it is designated as an *Out* variable and its value cannot be read inside the EFL-block. Likewise, if a variable is read inside an EFL-block, it is designated *In* and cannot be assigned to anywhere in the EFL-block. The EFL pre-compiler identifies the *In* and *Out* variables and notifies the programmer of any violations of this rule. It is clear that in the non-EFL parts of the program, which are completely sequential, and their semantics are deterministic, the values of variables can be changed without any restriction.

(c) *Once-only* assignments. Once a variable is designated as an *Out* variable, it does not make sense to assign to it multiple times in the same EFL block because the order of assignments is unknown and thus the final value of the variable is also unknown. Thus, multiple assignments to the same OUT variable are prohibited in EFL.

In an earlier paper [3] we used Rice's theorem [5] to show that violations of once-only assignment cannot be checked at compile

time. We therefore implemented a debug mode in which the EFL pre-compiler produces code in the host language, that checks for violations of once-only assignments at runtime. In the long term we would like to find a complete and efficient implementation of our flexible computation approach to parallel programming and this is discussed further in [3]. (Further discussion on Rice's theorem and the possibility of circumventing it is to be found in section 5).In [3] we also validated EFL usability by a case study that addresses an important kind of parallel programs, which are referred to as *task trees* [7] [8] [9].

Comment on EFL's Programming Model Design Decisions. Different languages define functions and variables in different ways, a difference which is particularly felt when we compare typed versus un-typed languages. Typed languages require variables, functions and functions' parameters to be bound to data types. On the other hand, un-typed languages don't have those requirements. We wanted EFL code to exist within the context of the host language, with full access to the host program variables, context, and scope, and simultaneously to be host-language-independent, enabling EFL code portability among host languages, without any modification. In order to achieve those goals, variables and functions are not allowed to be defined inside EFL blocks, enabling EFL code to be embeddable in any host language, typed and un-typed alike

3 EFL GRAMMAR AND SEMANTICS

The resemblance between EFL's and C's syntax is apparent. For example, the following EFL code calculates the factorial of a collection of numbers:

```
for (i = 0; i < len(inSeq); i = i + 1)
{
   outSeq[i] = factorial(inSeq[i]);
}
```

This code would compile in C, C++, Java, C# and Javascript and it would do the same thing in each of those languages. A programmer familiar with the sequential execution model of any of these C-syntax-based languages can tell that the semantics of this for-loop is to iterate over the *in*Seq collection by incrementing *i* by one at the end of each iteration instance. The *factorial* function is called in each iteration instance with the current item of *in*Seq. The value returned by *factorial* is stored in the outSeq[i]. The EFL semantics of the same *for*-loop, according to its flexible order-independent execution model, is fundamentally different: instead of looping through each iteration instance of the for-loop, sequentially, one at a time, proceeding to each subsequent iteration instance only after finishing the body of the current iteration instance, EFL executes each iteration instance of the loop in *parallel*. Each of the calls to the *factorial* function is executed in parallel, as different and separate tasks. If the computer executing the program has multiple cores, they are utilized in the most efficient way.

Basic Parallel Constructs

The basic constructs of EFL are:

(a) Assignment-block:
Similar to C, it is a set of assignment expressions, with variable(s) being assigned to, followed by the assignment operator (the equals sign), followed by the expression whose value is assigned to the variable(s). For example:

```
EFL{
myValue = 5;     // Simple assignment of value
myExpr = f(5);  // Expression containing function call
}EFL
```

Whereas in a sequential execution model language, these two statements would execute in sequence, according to their occurrence in the text of the program, in EFL they are executed in an unspecified order; actually, in parallel. The benefit of this is particularly felt when used with calls to functions that are CPU-intensive. For example:

```
EFL{
myVal1 =  cpuIntensiveOp(someParameter);
myVal2 =  cpuIntensiveOp(someOtherParameter);
}EFL
```

This example shows EFL's Assignment-block's core semantics: the two calls to the long-running function cpuIntensiveOp are executed in parallel and their returned values are assigned to the OUT variables myVal1 and myVal2. An OUT variable cannot be used as index to an array arr,but the value returned by a function call may be used as index (for example arr[f(x)]). In this case f(x) is evaluated in parallel with other statements in the EFL-block. Likewise, if a function is called with values which are a result of function calls, then those are calculated in parallel but *not* the 'outer' function call. Note that the evaluation of such expressions in EFL is similar to the substitution model of expression evaluation in functional programming languages like Scheme [10]. In the following assignment statement, the EFL pre-compiler generates code in which all the calls to f are calculated in parallel and only after the calls to f return their results, is g called.:

```
arr[f(a)] = g(f(x) * f(y), f(z));
```

(b) *if*-block:
This construct allows conditional execution of code. It allows for alternative conditions using the keyword elseif, and a default condition else that is executed when none of the conditions result in true. A typical *if* block can look like the following:

```
EFL{
if ( conditionOne(someValue) ) {
   // some code here
} elseif ( conditionTwo(someValue) ) {
   //other code here
} elseif ( conditionThree(someValue) ) {
   //other code here
} elseif ( conditionFour(someValue) ) {
   //other code here
} else {
   //last code here
}
}EFL
```

In terms of functional programming, the if statement in EFL is similar to a Special Form. Although function calls in EFL are usually executed in parallel, function calls in the condition of an if statement are not executed in parallel, but sequentially, as usual in sequential execution languages. That way, the body of the block controlled by the first true condition is executed in parallel to the rest of the EFL-block. If the conditions were evaluated in parallel to the rest of the *if*-block (like they are in pif, see below), the decision of which block to execute inside the if statement would have to be delayed until the results of all the conditions are determined. This would create a barrier in the parallel execution of the whole EFL-block. Therefore, we decided that the conditions of an if statement are not evaluated in parallel. If the programmer would like to utilize parallelization to evaluate if conditions, then he has two options: (1) to use a pif-block (see below), or (2) to write an EFL-block in which the necessary conditions are evaluated in parallel and the results are saved to variables, and then, use the results in a subsequent EFL-block. This could look like the following:

```
EFL{
a = f(x);
b = g(x);
}EFL
EFL{
if (a) {
   //some code
} elseif (b) {
   //some more code
} else {
   //other code
}
}EFL
```

That way, the programmer has more control over the conditions. It is important to note that this is not a violation of the *In* and *Out* variables rule. When a and b are assigned in the first EFL-block, they are established as *Out* variables. However, that only applies to the scope of the first EFL-block. In the next EFL-block, their *In* / *Out* scope is reset and when they are encountered there, their values are read, thus defining them as *In* variables. This is perfectly valid and in fact encouraged in EFL.

(c) *pif*-block:

It is similar to the *if block* in that it provides a mechanism for conditionally executing code. Using the keyword pif, this construct allows for all of the conditions of the pif to be evaluated in parallel. The pif computes the conditions of a pif elseif else in parallel so that by the time all of the conditions' values are calculated, the block controlled by the first true condition could be jumped to and executed. There are certain issues that must be taken into account with this approach. Take for example the following code:

```
EFL{
pif (a == 0) {
   //something
} elseif ((b / a) == c) {//possible divide by zero exception
   //something else
}
}EFL
```

In a sequential language, there would be no chance of a 'divide by zero' exception when executing this code. In a parallel execution, when all conditions are evaluated in parallel, then regardless of the result of the first condition, the second condition would also be evaluated. If in fact the value of a is zero, this would result in a run-time error. One downside of the pif is that it adds a blocking phase until all the conditions of the pif-block are evaluated.

(d) *for*-block:

It is a loop construct allowing for a block of code to be executed multiple times. Due to the once-only assignment restriction, *for* blocks are only useful when used to loop through any sequence data type. Each iteration instance must save any calculated values to a different slot in a sequence. That way, there will be a well-defined result at the end. For example:

```
EFL{
for (i = 0; i < 10; i = i + 1) {
   mySeq[i] = cpuIntensiveCalculation(i);
}
}EFL
```

By assigning to different locations of the sequence in parallel, we can ensure deterministic results. Assigning to a different location of the sequence in each iteration instance avoids violations of *once-only assignment*. As a side note, the variable i violates both the *In* and *Out* variable rule and *once-only assignment*. It is assigned in the initialization of the loop, defining it as an *In* variable. After the body of the loop executes, the value of i is incremented, violating *once-only assignment*. Its value is subsequently read in the condition of the loop to decide if the loop should continue to the next iteration instance or not. Reading the value of i defines it as an *In* variable. Having already been defined as an *In* variable, i violates the *In/Out* designation rule. Because those rules are a safeguard against non-deterministic results of a parallel execution, and the variable i is never actually assigned to or read in parallel in the processed code in the host language, this exception to EFL's programming model was allowed. It was decided that for the benefit of the programmer, the *for*-block syntax remains similar to the C for-block syntax.

Advanced Parallel Constructs

Although EFL was designed to mirror a sequential language as closely as possible, there were some advanced parallel constructs that were included in the language:

(a) *LogLoop*-block:

It was created in order to allow performing an efficient parallel scan of a sequence. This parallel construct is implemented by a parallel algorithm that executes an operation on non-overlapping sub-sequences of a sequence.

A call to *LogLoop* looks like this:

```
EFL{
result = logloop(mySeq, myFunction);
}EFL
```

EFL's *LogLoop* construct is implemented using an algorithmic pattern which is based on balanced trees. The idea is to build a balanced tree on the input data and traverse it from the leaves to the root. A k-ary tree with n leaves has $log_k(n)$ levels and therefore the running time of the algorithm is $O(log_k(n))$ where k is the number of parameters of the function, and n is the length of the sequence. One example function is addition. The algorithm works by summing every two elements in parallel, then summing every two resulting values and so on. This will produce the sum of all values of the sequence. The *LogLoop Block* requires the programmer to specify the function to apply and the sequence on which to apply the function.

The number of input elements for the function is dynamically detected at run-time and the algorithm acts accordingly. If the function accepts three parameters, then the running time of the algorithm is $O(log_3(n))$. The following pseudo-code illustrates the execution in the background:

```
i = 0
for h = 1 to log2(n) {
  i = h
  while (i < = (n / 2^h)) {
    z = Seq[i] op Seq[i+1]
    Seq[i+1] = z
    i += 2
  }
}
```

It should be emphasized that the *LogLoop* Block is the most appropriate EFL tool to implement the Reduce Parallel Design Pattern, which implementation is described in section 4 below.

(b) *Loop*-block:

Whereas traditional loops like while and for work sequentially, EFL executes loops in parallel. These traditional loops are syntactic sugar around a sequential execution of code with a goto statement at the end of the statements of the loop body. A while loop in C is equivalent to:

```
BEGIN_LOOP:
   if (not(condition)) {
      goto END_LOOP;
   }
   //statements of the loop
   goto BEGIN_LOOP;
END_LOOP:
```

In EFL, instead, the *Loop Block* was created as an abstraction of loops that decouples the repetitive nature of a loop from the order of execution. Here is an example:

```
:loop (i = 0)
{
   if (i < len(seqIn)) {
      loop(i + 1);
   }
   seqOut[i] = cpuIntensiveFunc(seqIn);
}
```

The *Loop Block* is declared with an identifier and an initial value for the iteration variable. That identifier (loop in the previous example) is defined by the programmer, and is not a language reserved keyword. Within the body, a recursive call is made to the loop using the identifier, and passing a new value for the *new* iteration variable. This provides the ability to advance the iteration using varying increments of the iteration variable. The *Loop Block* is "syntactic sugar" for a function declaration, and is looped through using a recursion-like execution. This style of programming is especially suited for parallel programming [11] where all iterations of the loop are executed in parallel. The loop block views the body of the loop having its own local variables for each index value of the loop; so, all the iterations may be executed in parallel.

(c) *MapLoop*-block:

This construct is based on the "map" function of functional programming [10] [12]. It applies a function to every element of a collection. Calculations are made on every element of the sequence and results are saved into a new sequence, not in the original one. This is trivial to parallelize because the execution of the function with a given element in the collection is independent of the execution of the function with a different element in that collection, and order of execution is irrelevant. For example:

```
EFL{
   squares = maploop(mySeq, square);
}EFL
```

Although in general EFL was designed to look like a sequential imperative language like C or C++, the *MapLoop* was included because it helps encourage thinking about loops in a non-sequential way.

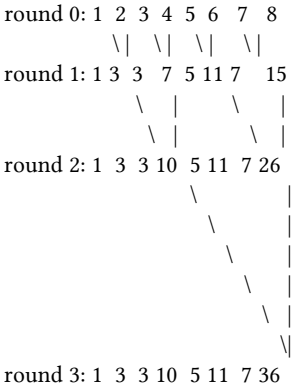
4 PARALLEL PROGRAMMING PATTERNS IN EFL

A Design Pattern describes a standard solution of a well-known class of design problems. It also suggests how to approach this kind of problems. We have implemented several parallel design patterns [14] using EFL as proof-of-concept project. A full report of this project will be published soon. One of the cases implemented in that project will be presented in the following paragraph: let's use EFL's *LogLoop* construct to implement the Reduce pattern.

Assume we need to calculate the sum of the integers 1 through 8:

```
def mySum(L):
   func = int.__add__
   EFL{
   result = logloop(L, func)
   }EFL
   return result
print(mySum([1,2,3,4,5,6,7,8]))
```

A graphical view of the algorithm's execution looks like this:



At the end of round 3 ($log_2(8) = 3$), we have the result 36.

5 CONSEQUENCES OF RICE'S THEOREM REGARDING PARALLEL COMPUTATION

In our earlier paper [3], we used Rice's theorem to show that violations of once-only assignment cannot be checked at compile time. Here we generalize and present various restrictive results which are consequences of Rice's theorem. Then we suggest an approach for circumventing these restrictions.

Rice's Theorem: Let C be the set of computable functions of n arguments and let X, Y be non-empty subsets of C such that X∪Y=C, X∩Y≠Ø. Then the problem of determining whether the function computed by a program is in X or in Y is undecidable. Rices's theorem concerns the functions computed by programs. The following Rice-like theorem is a consequence, and concerns certain behaviours of parallel executions.

Rice-like theorem regarding certain behaviours of parallel execution

Consider any program behaviour B of parallel execution which does NOT occur with sequential executions - e.g. concurrent multiple assignment (or race condition), deadlock, etc. Suppose too that B can occur when executing some parallel program and does not occur when executing some other parallel program. Then the problem of determining whether or not this behaviour may occur when executing an arbitrary parallel program is undecidable.

Proof: Suppose to the contrary that this is decidable. By the above assumption, there exists a parallel program P which exhibits this behaviour and some other program Q which does not exhibit this behaviour. Let f(x) be any computable function and suppose that f(x) is computed sequentially. Then clearly this behaviour does not occur in the computation of f. Consider the following program:

if f(x)=0 then P else Q.

In the above program, the behaviour B occurs iff f(x) is sometimes zero. Thus, if we had a decision procedure for this behaviour we would have a decision procedure for testing whether or not f(x), implemented sequentially, is sometimes zero.

This is impossible by Rice's theorem. Thus, this problem is undecidable.

Let us now consider a different problem regarding certain behaviours of parallel execution.

A different problem

Let B be some behaviour as above concerning parallel execution. Let Z be a nonempty set of programs, where Z does not include all programs.

NOTE: Parallel programs are included.

Then deciding whether or not this behaviour may occur when executing an arbitrary program where an incomplete decision procedure gives a don't know answer on Z is a possibility not excluded by the Rice-like theorem above. In particular, the following holds.

A theorem regarding this different problem

a) Such a Z exists for any such behaviour B.

b) Z must be infinite.

c) No such minimal set Z exists.

Proof: Clearly there is a program P which exhibits this behaviour and a program Q which does not exhibit this behaviour. Let R be any program.

a) Let Z be the set of computable programs except for P, Q.

Consider the following.

if R=P
then Behaviour B occurs
elsif R=Q
then Behaviour B does not occurs
else don't know

This decides whether or not R has behaviour B and gives a don't know answer on Z as required.

b) Let E be a program for testing whether or not a program may exhibit behaviour B, where it gives a don't know answer on Z. If Z may be written as the union of Z1 and Z2 where for programs on Z1, the behaviour B occurs in some execution. For programs in Z2, the behaviour B never occurs. Z1, Z2 are of course disjoint. Now, if Z is finite, then Z1, Z2 are also finite as they are subsets of Z.

Now consider the following.

if R ∈ Z1
then Behaviour B occurs in some execution
else if R ∈ Z2
then Behaviour B does never occurs
else Run the program E to decide the outcome.

This would mean that we would have a decision procedure for the Rice-like theorem above, which is impossible.

Thus, Z is infinite.

NOTE: Only a finite number of textual comparisons need to be made in the tests R ∈ Z1, R ∈ Z2.

c) Similarly, there is no minimal set Z, since for any such set Z, then Z \ {P} is a smaller set satisfying this theorem since Z \ {P} cannot be empty as this would mean that we would have a

decision procedure for the Rice like theorem above which is impossible.

Is there a way around the restrictions of Rice's theorem and its consequences?
We have seen that Rice's theorem and its consequences severely restrict what can be checked at compile time. The previous results show that problems associated with the behavior of parallel execution are undecidable and thus cannot be handled completely at compile time (a-priori). Thus all compile time solutions to behavioural problems of parallel execution will be incomplete and it seems that complete solutions must handle (parts of) these problems at run time. Nonetheless, there may be a way around.

Example 1: desirable property = freedom from conflict
Purely recursive programs with no assignment statements are free from conflict and every computable function can be computed by such a function.

Example 2: desirable property = more efficient
For sequential assembly programs, a compile time check can be made that we do not store a register to a memory address and then load a register from the same address. Doing this does not affect the set of computable functions.

Rice's theorem does NOT apply at all in such cases since we do not divide the class of computable functions into two non-empty disjoint classes.

Specifically, can we devise a compile time mechanism based on the above, for freedom from conflict with more efficiency in some sense?

More precisely, is there a decidable subset of the set of all programs such that:
a) They compute all the computable functions.
b) They are free from conflict.
c) For every program not in the subset, there is an equivalent program in the subset which is at least as efficient.

We do not know if such a set of programs exists. Furthermore, it may be that (c) is impossible, and if so, what kind of efficiency can be attained subject to (a) and (b)? Further work is needed to clarify this matter.

6 RELATED WORK

Many languages and frameworks [6] (OpenMP, Erlang, Cilk, MPI, etc…) have been developed to support parallel programming. In OpenMP [6] [15] the programmers insert directives into their source code which tell the compiler which parts of the program should be parallelized (for example, #pragma omp parallel). The programming model of OpenMP is shared-memory-based. If variables are defined outside a parallel region or they have global scope, they are shared between threads. The fact that threads might update shared memory items creates a synchronization and locking problem. Such a problem does not exist in EFL because of the three principles of Flexible Computation presented above (only "pure" function calls, In and Out variables, Once-only assignments). OpenMP includes several synchronization constructs to manage the order of execution of instructions among a collection of threads whereas in EFL this is handled automatically by the code generated by the pre-compiler.
Cilk [16] is a C-based multithreaded language similar to OpenMP, based on a shared-memory model. Like EFL, its runtime system is responsible for thread scheduling. In addition to this , Cilk also deals with load balancing and inter-thread communications. A few keywords (cilk, spawn, sync) are provided for creating parallelism and handling synchronization.
Erlang [17] is a declarative language that shares many features with functional and logic languages. Each Erlang program starts as a purely sequential process and concurrency is dealt with by using several concurrency constructs. The Erlang memory model is not a shared memory model and asynchronous coordination between processes is achieved through explicit message passing. EFL does not explicitly support message passing at its programming model, but message passing may be used as a parallel platform upon which the EFL execution model may be implemented. It is worthwhile to note that an MPI-based version of the EFL pre-compiler is in the making.
Using the TPL extensions library for parallel programming [18], existing sequential code is potentially parallelized. In EFL, like in openMP and Cilk, it is the programmer's responsibility for determining which parts of the code should be parallelized. It is worth noting that in our implementation of EFL, the pre-compiler parallelizes function calls that occur inside EFL-blocks by using Python's parallel programming modules, freeing the programmer from dealing with those Python's modules directly (see our earlier paper [3]). In contrast to TPL's parallel *for*, in EFL there are no syntactic difference between the EFL for-block and its sequential sibling. EFL's *for*-block iterations are independent, however the sequence of values of the for index variable are computed sequentially in advance. The code generated using TPL adapts itself to the platform on which it runs, in a similar way to our Python's pools implementation of EFL's parallelism. According to [18], "TPL does not help to correctly synchronize parallel code that uses shared memory". Synchronization problems are avoided in EFL as we have previously explained. Similar difficulties will occur with other libraries which provide parallelism (e.g. MPI [6] [19]), since compile time checks are not possible with library implementations of parallelism.
Etsion et al. [20] describe out-of-order task pipeline execution. Their semantics definition is in terms of sequential execution order, while EFL's semantics definition is in terms of a flexible execution order. They argue that common task-based models require from the programmer to take care of inter-task data dependencies, which is very burdening. In EFL, instead, the parallel tasks created by the different EFL constructs are completely independent from each other; therefore, there are no inter-task data dependencies. .
Hoare's 1971 [21] and Lipton's 1975 [22] classic papers in the field of parallel programming describe techniques where formal logic

is used to check at compile time that a parallel program gives well defined results. For example, variables in a "parbegin ... parend block" may be both read as well as written. EFL's approach is different as we have explained above.

A related work of ours is about Flexible Algorithms and a course on this topic has been developed for complete beginners. Our teaching experiences are described in [23]. This course was well received and the full course notes are available at http://homedir.jct.ac.il/~rafi/flexalgo.pdf.

7 CONCLUSIONS

The EFL project's goal has been to create an Embedded Flexible Language whose foundations are based on an order-independent execution model. Code written in this kind of language may be embedded into *any* host language code. In its embedded blocks of code, execution may proceed in serial or in parallel order, having *deterministic* semantics.

The EFL language has an imperative programming style, promoting a small learning curve. This should simplify its adoption by programmers and teachers.

We hope that EFL will be found useful in scientific programming, mathematical libraries, etc, and as a teaching language to introduce parallel programming for beginners.

8 FURTHER WORK

We see a need for work in the following areas:

(a) How to weaken the EFL rules while retaining the determinism of the EFL unordered execution model.

(b) Handling of exceptions in the pif construct, which will allow the parallel execution of expressions which their Boolean conditions hold.

(c) Support for Purely Functional Data Structures [24]. Is this kind of data structures an alternative to the STM [25] model? May we integrate both in the EFL framework?

(d) Develop an EFL-based parallel programming curriculum.

(e) Clarify if there is a way around the restrictions of Rice's theorem and its consequences.

(f) Evaluate the ease of use of EFL with programmers.

ACKNOWLEDGMENTS

We wish to thank the Research Authority of the Jerusalem College of Technology for supporting the Flexible Computation Research Laboratory (FLEXCOMP Lab) where this research has been carried out. Web site: http://flexcomp.jct.ac.il/

We wish to thank J. J. Florentin for his invaluable assistance and encouragement.

Thanks to the referees for their helpful comments and suggestions.

REFERENCES

[1] A. Marowka. 2011. Back to Thin-Core Massively Parallel Processors. *IEEE Computer*, 44, 12 (Dec. 2011), 49-54. DOI: https://doi.org/10.1109/MC.2011.133

[2] R. B. Yehezkael, M. Goldstein, D. Dayan, S. Mizrahi. 2015. Flexible Algorithms: Enabling Well-defined Order-Independent Execution with an Imperative Programming Style. In *Proc. of the 4th Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC 2015). IEEE Press, 75-82.* DOI: https://doi.org/10.1109/ECBS-EERC.2015.20

[3] D. Dayan, M. Goldstein, M. Popovic, M. Rabin, D. Berlovitz, O. Berlovitz, E. Bosni-Levy, M. Neeman, M. Nagar, D. Soudry, R. B. Yehezkael. 2015. EFL: Implementing and Testing an Embedded Language Which Provides Safe and Efficient Parallel Execution. In *Proc. of the 4th Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC 2015), IEEE Press, 83-90.* DOI: https://doi.org/10.1109/ECBS-EERC.2015.21

[4] M. Chakravarty, F. Schröer and M. Simons. 1995. V-Nested parallelism in C. *In Proc.of Programming Models for Massively Parallel Computers, IEEE Press,167-174.* DOI: https://doi.org/10.1109/PMMPC.1995.504355

[5] H. Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. AMS 74, 2*, 358-366. DOI: https://doi.org/10.1090/S0002-9947-1953-0053041-6

[6] M. Sottile, T. Mattson and C. Rasmussen. 2010. *Introduction to Concurrency in Programming languages*, Chapman & Hall/CRC.

[7] M. Popovic, I. Basicevic and V. Vrtunski. 2009. A Task Tree Executor: New Runtime for Parallelized Legacy Software. In *Proc. 16th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems, IEEE Press, 41-47.*. DOI: https://doi.org/10.1109/ECBS.2009.11

[8] I. Basicevic, S. Jovanovic, B. Drapsin, M. Popovic and V. Vrtunski. 2009. An Approach to Parallelization of Legacy Software. In *Proc. 1st IEEE Eastern European Regional Conference on Engineering of Computer Based Systems (ECBS-EERC 2009),, IEEE Press*, 42-48. DOI: https://doi.org/10.1109/ECBS-EERC.2009.8

[9] M. Popovic and I. Basicevic. 2010. Test case generation for the task tree type of architecture. *Information and Software Technology*, 52, 6, 697-706. DOI: https://doi.org/10.1016/j.infsof.2010.03.001

[10] P. Hudak. 1989. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21, 3, 359-411. DOI: https://doi.org/10.1145/72551.72554

[11] T. Timothy Mattson, B. Beverly Sanders and B. Massingill. 2004. In *Patterns for parallel programming*, Addison-Wesley Professional.

[12] H. H. Abelson and G. Sussman. 1996. *Structure and Interpretation of Computer Programs*, 2nd ed., MIT Press, 1996.

[13] K. Batcher. 1968. Sorting networks and their applications. In *Proc. of the AFIPS Spring Joint Computer Conference*, ACM, 307-314. DOI: https://doi.org/10.1145/1468075.1468121

[14] D. Dayan, M.Goldstein, E.Bussani Levy, M. Naaman, M. Nagar, D. Soudry, R. B. Yehezkael. 2016. Implementing Parallel Programming Design Patterns using EFL for Python. *Oral Presentation at the EuroPython 2016 Conference*, Bilbao, Spain.

[15] OpenMP web site http://www.openmp.org/ .

[16] Cilk web site http://supertech.csail.mit.edu/cilk/ .

[17] Erlang web site http://www.erlang.org/ .

[18] D. D. Leijen and J. Hall. 2007 Parallel Performance: Optimized Managed Code For Multi-Core Machines, *MSDN Magazine* (Oct 2007).

[19] http://www.open-mpi.org/.

[20] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. Badia, E. Ayguade, J. Labarta and M. Valero. 2010. Task Superscalar: An Out-of-Order Task Pipeline. In *Proc. of the IEEE/ACM Intl. Symp. on Microarchitecture(MICRO-43)*, IEEE Press, 89-100. DOI: https://doi.org/10.1109/MICRO.2010.13

[21] C. A. R. Hoare. 1972. Towards a theory of parallel programming. *Operating Systems Techniques*, 9, 61-71.

[22] R. J. Lipton. 1975. Reduction: a method of proving properties of parallel programs. *Commun. ACM, 18, 12*, (Dec. 1975),717-721. DOI: https://doi.org/10.1145/361227.361234

[23] R. B. Yehezkael 2006. Flexible Algorithms: Overview of a Beginners' Course. In *IEEE Distributed Systems Online, vol. 7, no. 11*, art. no. 0611-oy002. DOI: https://doi.org/10.1109/MDSO.2006.65

[24] Ch. Okasaki. 1999. *Purely Functional Data Structures*, Cambridge University Press.

[25] M. Herlihy and N. Shavit. 2008. *The Art of Multiprocessing Programming, Burlington*, Morgan Kaufmann.