MORTAL - Multiparadigm Optimizing Retargetable Transdisciplinary Abstraction Language

Ove Kåven[1,2] and Lars Ailo Bongo[2]
[1]Kongsberg Spacetec AS, Tromsø, Norway
[2]Department of Computer Science, University of Tromsø, Norway
ovek@arcticnet.no, larsab@cs.uit.no

Abstract
This short paper describes MORTAL, a new general-purpose programming language and compiler for high-performance scientific applications. MORTAL aims to bridge the knowledge gap between computer scientists and scientists by offering a multiparadigm programming environment that allows connecting the mathematical formulae written by scientist to algorithms implemented by the software engineer in a natural way, and understood by both. We provide the rationale for MORTAL, give an overview of the language design and the MORTAL compiler. The compiler is self-hosting, and our initial evaluation shows that MORTAL programs have similar performance as C programs.

1. Introduction

Computational modeling and analysis has become essential in many scientific and engineering disciplines. However, to solve current problems in computational science ever more powerful software and hardware is required. Existing high-performance programs have shown their usefulness, but efficient implementations typically require trained computer scientists. Unfortunately, a computer scientist may not have the necessary mathematical or statistical background to understand the computational problem to be solved, and may therefore miss potential mathematical transformations of the problem that might increase numerical accuracy or reduce computation time. In addition, experienced computer scientists are usually in short supply. A solution is therefore needed that makes existing computer scientists more efficient, reduces the need for them, or both.

We believe a novel programming language can solve both problems if it has the following features:

1. *Multiparadigm*: it must be possible for the language to be used effectively by both software engineers through imperative or functional programming, and scientists through declarative programming. By having multiple paradigms in the same language, the mathematical formulae declared by the scientist can be connected to the computation frameworks and algorithm libraries implemented by the software engineer in a natural way, understood by both (as demonstrated in for example [1]).
2. *Optimizing*: the language should be designed for minimal overhead, and for compiling to optimized machine code that can take full advantage of the available hardware.
3. *Retargetable*: the language should make it possible to abstract away platform specifics without losing performance. It should be possible to allow the same source code to compile for various operating systems, CPUs, GPUs and even FPGAs. Ideally, the business logic should be independent of the software libraries used to perform the computations.

4. *Transdisciplinary*: the language should be general-purpose, and make it possible to write programs that can combine knowledge from different domains and be useful for different fields.
5. *Abstracting*: the language should be easy and intuitive to use, and automate away as many implementation details as possible, allowing the programmer to focus on the concepts that are really of concern.

We have investigated numerous programming languages, but to our knowledge, none solves all five requirements (we provide a detailed discussion in [2]). In particular, the most relevant multiparadigm languages have the following issues: Nim [3] is too low-level, Rust [4] is too restrictive, Scala [5] is JVM based, Wolfram [6] is proprietary and dynamic, Oz [7] require its own VM, and Julia [1] is dynamic.

We propose a language called MORTAL that achieves all five goals. To reduce the complexity of the language, we take full advantage of modern compiler technology, and provide strong metaprogramming facilities. MORTAL also provides declarative programming, and contract-based programming [8]. The above mulitparadigm languages, and several other languages inspired the design of MORTAL. Its syntax is from Pascal, Java, Python, and C/C++/C#, and it provides mathematical operations as in Matlab. Objective-C and Vala [9] inspired the MORTAL memory management.

The following section gives an overview of MORTAL. Section 3 provides an initial evaluation, and Section 4 concludes.

2. MORTAL Language and Compiler Design

MORTAL is a new metaprogrammable language with a syntax designed for use in high-performance applications (a detailed description of the design and syntax is in [2]).

To interoperate with existing libraries and frameworks, we have initially implemented the traditional procedural and object-oriented programming paradigms, with a few adaptations for metaprogrammability, automatic memory management, performance, and productivity. We have not yet designed the corresponding declarative paradigms, but the syntax is flexible enough to integrate these later. This partially satisfies the *Multiparadigm* and *Retargetable* requirements.

To make MORTAL customizable and usable in different domains, it provides a number of overloadable operators and sufficient metaprogramming power to grant a natural syntax to the use of external libraries and frameworks. This satisfies the *Transdisciplinary* and *Abstracting* requirements.

To make it run at high performance on any available hardware, MORTAL is a compiled language that also makes provisions for runtime code generation and runtime algorithm specialization. This satisfies the *Optimizing* requirement.

To make MORTAL easy to learn for programmers familiar with other languages, it uses a classic curly-braces syntax, with a few adaptations to make the syntax ambiguity-free.

We have designed and implemented an optimizing compiler for MORTAL (figure 1). It implements a large part of the language design, and some of the memory management. The compiler is written in its own language, and compiles itself (self-hosting). The compiler generates C code, which is then compiled to machine code. Generating C code is an easy and popular way to make code generator backends for high-level languages, but we also plan to implement other code generators to satisfy the *Retargetable* requirement. In addition, MORTAL is compatible with libraries such as Glib, and we intend to interface with frameworks such as Spark [10].

3. Initial Evaluation

The implementation of the MORTAL language is not yet complete, but we can evaluate the current implementation with respect to correctness, usability, and performance.

3.1. Correctness and Usability

Currently the biggest test for correctness is whether the MORTAL compiler is able to compile itself correctly. To test this, we compile the compiler thrice. First, we build a reference compiler from the C sources in version control. We use the reference compiler to build a test compiler, and the use the test compiler to build itself. The resulting compiler must compile without errors and pass the MORTAL unit test-suite, which the current version does.

Since we have written the MORTAL in MORTAL, we also believe its implementation show the usability of the language. Especially, we have found MORTAL's run-time type information (RTTI) system and its multimethods very useful in the construction of the compiler. Multimethods have allowed manipulating the Abstract Syntax Tree (AST) with an ease comparable to that of functional languages.

3.2. Performance

MORTAL aims to provide performance comparable to C/C++. To test this, we implemented the *fasta* benchmark from the Computer Languages Benchmark Game [11] in MORTAL and compared its performance with *fasta.gcc*, a reasonably well-optimized C implementation (a faster implementation exists, but it uses UNIX file descriptors and its own buffering, instead of standard I/O used by MORTAL)

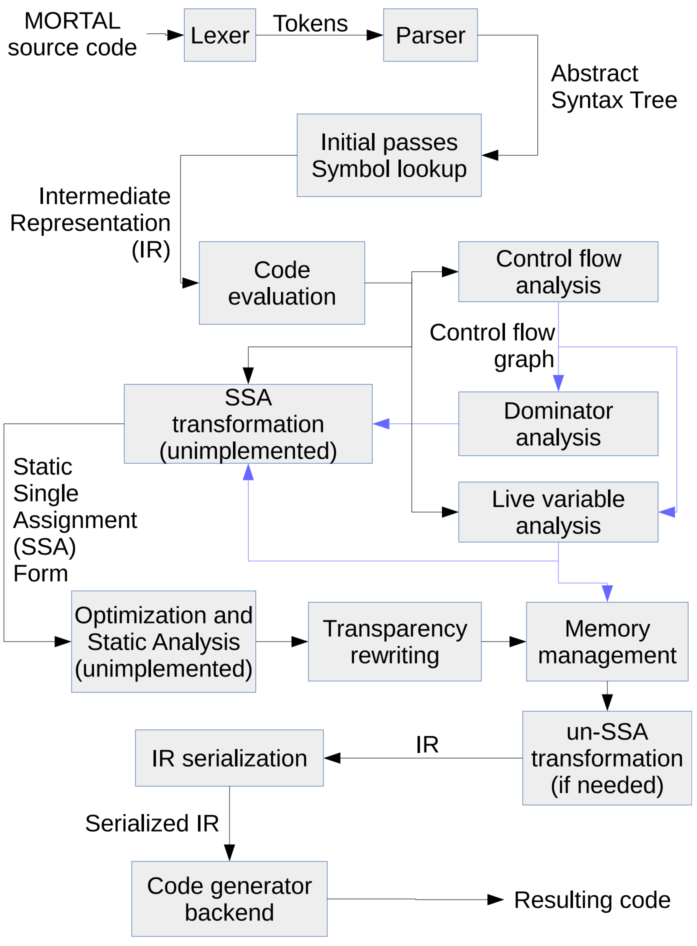

Figure 1 MORTAL compiler design.

We implemented two versions of the benchmark in MORTAL: without I/O optimizations, and with a level of I/O optimization comparable to the C version. The latter uses MORTAL's C compatibility features to do direct pointer manipulation, since we have not yet implemented slices in MORTAL.

Version	Average (sec)
Optimized C	3.219
Simple MORTAL	4.427
Optimized MORTAL	3.215

Table 1 Performance of fasta benchmark, with n = 10.000.000. Average over six executions.

The measurements were done on a laptop with an AMD A10-5757M quad-core CPU running at 2.5GHz, and 8 GB of RAM, running Debian jessie (gcc version 4.9.2). 2. The results show that a MORTAL program can achieve the performance of C (Table 1). The difference between MORTAL and C is mostly because of minor implementation differences between the C and MORTAL programs. Since MORTAL's compiler currently generates C code itself, it would otherwise not be possible for MORTAL to outperform well-written C code.

4. Conclusion

This short paper has introduced and described MORTAL, a new metaprogrammable programming language for high-performance applications, and its compiler. The language currently has procedural and object-oriented programming, and provides many important features such as RTTI, function and operator overloading, subtype and parametric polymorphism, multimethods, and exceptions. The language design satisfies most of its original goals, but we still need to design and integrate the declarative paradigms, and implement other code generators to satisfy all five.

The compiler is self-hosting and able to compile itself, showing that the language and its compiler is already usable. MORTAL programs match the performance of C programs. We believe that as MORTAL and its compiler matures, it will become a useful language for solving many of the demanding computational tasks in modern science, and for our future work in programming language research.

MORTAL is open source, released under the MIT license, and available from https://sourceforge.net/projects/

References

[1] Bezanson, Jeff, et al. "Julia: A fast dynamic language for technical computing." *arXiv preprint arXiv:1209.5145* (2012).

[2] Ove Henrik Kåven. *Multiparadigm optimizing retargetable transdisciplinary abstraction languag*e. Master's thesis, Dept. of Computer Science, University of Tromsø. April 2015. http://hdl.handle.net/10037/7730

[3] Nim Programming Language. http://nim-lang.org/.

[4] The Rust Programming Language. http://www.rust-lang.org/.

[5] Odersky, Martin, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *An overview of the Scala programming language*. No. LAMP-REPORT-2004-006. 2004.

[6] Wolfram Language. https://www.wolfram.com/language/.

[7] Smolka, Gert. *The Oz programming model*. Springer Berlin Heidelberg, 1995.

[8] B. Meyer, *Applying 'design by contract*, Computer, vol. 25, pp. 40--51, Oct 1992.

[9] Vala - Compiler for the GObject type system, https://wiki.gnome.org/Projects/Vala.

[10] Zaharia, Matei, et al. "Spark: cluster computing with working sets." *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Vol. 10. 2010.

[11] Computer Languages Benchmark Game. http://benchmarksgame.alioth.debian.org