The expression problem in platform-aware programming

Francisco Heron de Carvalho Junior
heron@dc.ufc.br
Departamento de Computação, Universidade Federal do Ceará
Fortaleza, Brazil

Abstract

Platform-aware programming involves making assumptions about specific features of the target execution environment to improve performance. We restate a fundamental problem in software extensibility called The Expression Problem for platform-aware programming to show how a solution based on dynamic multiple dispatch over platform types may improve such a performance engineering practice.

1 Introduction

Architectural independence is a basic design premise of programming languages. It promotes high-level abstractions and delegates to compilers the problem of running a program on multiple computer architectures, contributing to dealing with the increasing complexity of software in the face of the rapid evolution of hardware. Architectural independence was possible due to the Von-Neumann architecture and industry efforts to consolidate standards for designing hardware components. However, architectural independence constitutes a performance bottleneck in many high-performance computing (HPC) applications.

HPC emerged in the 1970s with the first supercomputers, today represented by clusters and MPPs[1] [6]. HPC system designers have introduced several features to build ever-faster parallel computers, including vector processing, multiprocessors with deep memory hierarchies, low-latency interconnections for distributed-memory parallelism, and heterogeneous computing. Programming for HPC systems became a task for *performance engineering* experts as architectural dependence proved necessary to exploit their performance.

In recent years, performance engineering concerns have reached application niches of wider interest, such as Artificial Intelligence (AI). Most of the success of machine learning through Deep Neural Networks (DNNs) is devoted to GPUs. Distributed computing also plays a key role in the most challenging deep learning tasks, and the first supercomputers aimed at AI have been deployed, such as Intel's Aurora, Microsoft Azure's Eagle, and NVIDIA DGX SuperPOD [6]. In HPC research, *heterogeneous computing* challenges and the HPC/AI convergence dominate debates. In the hardware industry, the most relevant HPC players are working on designing AI processors and accelerators, as well as quantum computing devices, drawing attention to the Von-Neumann bottlenecks and motivating predictions that its predominance is coming to an end [9].

We argue that the improvement of programming practices for HPC has been hampered by strict assumptions regarding architectural independence in the design of programming languages, which leads to a concentration of efforts on high-level abstractions and compilation techniques to hide the natural architectural heterogeneity of HPC systems. In turn, efforts to improve the practice of *platform-aware programming*, i.e., writing code that makes assumptions about features of the target execution environment, are rare. However, the relevance of platform-aware programming tends to increase as computer system designers continue to develop new accelerator architectures and instruction set extensions for general-purpose processors to address specific application requirements. So, we draw the programming language community's attention to the problem of improving high-level programming languages for platform-aware programming by looking for structured ways of writing platform-aware code without performance losses and satisfying software modularity and extensibility requirements.

This paper contributes to this issue by proposing, in Section 2, a restatement of the *expression problem*, a well-known problem in programming language design, for platform-aware programming. Then, in Section 3, it presents an attempt to address this problem in the Julia programming language [3] based on *multiple dispatch* over *platform types*. Finally, Section 4 discusses the limitations and drawbacks of this approach and lines for further work.

2 Platform-aware expression problem

In a post on the Java Genericity mailing list in 1998, Phil Wadler coined the term *expression problem* to refer to a fundamental problem in software extensibility [2, 11]. How programming languages solve the expression problem is a measure of their expressiveness. We restate the expression problem for platform-aware programming by adding the notion of *platform* to *data types* and *operations*.

Let $\mathcal{D}$ be a set of *cases* in the definition of a variant data type and $\mathcal{O}$ be a set of *operations* over them, i.e., each operation has a definition for each case. By taking cases as rows and operations as columns of a decomposition matrix, the original expression problem concerns how a programming language can support adding new cases (rows) and

[1] *Massive parallel processors.*

operations (columns) without recompiling existing code and preserving static type safety, i.e., without dynamic casts.

W. Cook has identified two basic decompositions: procedural data abstraction (PDA), based on type cases (rows), and abstract data types (ADT), based on operations (columns) [2]. Object-oriented and functional programming languages are the most influential representatives of these two decompositions, respectively, and, for this reason, they will be used as references in the following discussion.

In a functional program, an operation (function), specified by scrutinizing the cases of a data type, can be added without modifying other functions or the data type definition. This is not true when adding a new type case, potentially requiring modifying all the functions over the type. In an object-oriented program, a new class specifies a new type case and the definition of the operations for it (methods). However, adding a new operation requires modifying all the classes that define each data type case to add the new operation. The literature describes many solutions to the extensibility limitations of functional and object-oriented languages regarding the expression problem (e.g., [5, 8, 10, 12, 13]).

For the restatement of the expression problem for platform-aware programming, we add $\mathcal{P}$, a set of platforms, so that, to describe a platform-aware program comprising a set of operations over the cases a data type to run efficiently on a set of target *execution platforms*, we now have a three-dimensional decomposition grid, with *data type cases* in the x-axis, *operations* in the y-axis, and *platforms* in the z-axis. In architecture-independent programming, $\mathcal{P}$ is useless since there is a single platform with minor assumptions about specific characteristics, if any, which compilers may handle.

A platform is defined as a set of features that specify characteristics that can be exploited in implementing operations to maximize its performance regarding some objective, such as reducing execution time, minimizing memory usage, and increasing power efficiency. For example, features specifying the presence and characteristics of GPU accelerators may be used to reduce execution time. Also, in programming for distributed-memory parallel computers, features that characterize the interconnection topology linking processing nodes may be used to minimize communication overhead and increase parallelism efficiency. However, platform-aware programming is not only for accessing hardware features through low-level APIs and system calls. For example, the choice between parallel algorithms having different scalability properties is driven by parallel computing platforms features like the number of processing elements and the performance of their interconnection [4].

As far as we know, no programming language supports a platform abstraction. Platform-aware programming relies on ad-hoc techniques to address platform assumptions statically, requiring recompilation for each target platform. This leads to the modularity and extensibility issues discussed below, isolated by assuming the original expression problem, i.e., for the x and y axis, is resolved.

In platform-based decomposition, a program comprises a set of co-existing versions for each target platform in separate modules. Assuming that the features of target computers may change over time, the correct platform version will be loaded at the program startup. This is a realistic assumption in programs running in containers or virtual machines in cloud-based environments, where the underlying execution platform may vary between executions. In such a decomposition, adding new operations or data type cases requires modifying all the platform versions. However, adding new platforms may also be problematic in practice because only some operations and data type cases need platform-specific implementations, and distinct platform features may be considered for different combinations of data type cases and operations. Therefore, platform-based decomposition tends to suffer from a large amount of architecture-independent code duplication over a large set of platforms based on a slightly different set of features.

In a functional language or anyone based on functional decomposition, *kernel functions* are a small subset of the functions in a platform-aware program that make assumptions about execution platform features, are aware of or detect them, and behave accordingly. Adding a new platform or changing platform assumptions (e.g., to exploit features of a new accelerator) could be problematic, as it requires modifying the code of all kernel functions.

Object-oriented languages suffer from analogous issues, requiring modifications to all subclasses representing data type cases if a new platform is added or platform assumptions are modified. This scenario tends to become more problematic because, in platform-aware programs, platform assumptions are more commonly applied to implementing operations than to implementing data type cases. While changes are restricted to a single subclass in the latter case, they are required across all subclasses in the former.

3 Case study: platform types for Julia

We have introduced the PlatformAware.jl package [3] to the ecosystem of the Julia programming language [1], aimed at *structured platform-aware programming*. There are three reasons for having adopted Julia. First, it facilitates the implementation of language extensions through metaprogramming. Second, it addresses HPC requirements by promoting performance engineering practices that achieve performance comparable to C and Fortran. Third, it supports a rich type system to enable dynamic multiple dispatch [7].

PlatformAware.jl uses dynamic multiple dispatch over *platform types* to address the platform-aware expression problem in Julia. Platform types are formally distinguished from data types used to implement a platform abstraction. They encode platform assumptions and features using Julia

abstract types, with subtyping relations defining specialization/generalization between them.

The following code presents signatures of two methods for a convolution function called imfilter, with regular parameters *img* and *kern*, making assumptions about features of the target execution platform through *platform parameters*:

```
@platform aware function imfilter({ accelerator_count::@atleast(2, A),
                                    accelerator_architecture::Hopper },
                                  img, kern) where A
@platform aware function imfilter({ node_count::@atleast(8, N),
                                    processor_core_count::@between(16,32, C),
                                    processor_simd::AVX512,
                                    accelerator_count::@just(0) },
                                  img, kern) where A
```

The first method will be selected when at least two NVIDIA Hopper GPUs are available, while the second one will be selected in a cluster with at least eight compute nodes, each having between at least 16 and at most 32 cores, processors supporting the AVX512 ISA extension, and no accelerators. The variables *A*, *N*, and *C* refer to each method's actual quantities of GPUs, nodes, and cores.

Thus, to work with PlatformAware.jl, the developer must first identify kernel functions, like *imfilter*, and write their platform-aware methods by using the **@platform aware** macro. Platform-specific assumptions are encoded in the method signature of each kernel method through the *platform parameters* (delimited by brackets), each typed by an assumption type. The platform parameters will define the *platform type* of the kernel method, which defines the method's target platform. They are chosen from a set of pre-defined platform parameters currently supported by PlatformAware.jl.

Each kernel function requires a fallback method free of platform parameters. It is called if there is no kernel method whose platform type is a supertype of the execution platform's platform type. They are declared using **@platform default**, as follows (only signature):

```
@platform default function imfilter(img, kern)
```

While the program supplies data-typed parameters, platform-typed parameters are supplied by the execution platform. So, kernel function calls ignore platform parameters. The **@platform** macro implicitly passes the *platform arguments*, which encode the actual features of the execution platform. For that, **@platform aware** rewrites the signature of kernel methods to add all platform parameters, and **@platform default** creates the *entry method*, with only data-typed parameters and making a call to the kernel function by passing the platform arguments. The kernel method whose platform type best meets the platform type of the execution platform, encoded in platform arguments, is selected by multiple dispatch.

Ideally, platform arguments should be dynamically calculated through a *feature detection* mechanism embedded in the language runtime. However, this is challenging, requiring non-portable tools outside the language environment. Also, the lack of standard conventions and APIs for identifying and classifying processors and accelerators from different vendors makes it hard to specify a hierarchy of platform types. For these reasons, PlatformAware.jl adopts a static approach where platform arguments are described in a configuration file called Platform.toml. It is manually editable by users and possibly provided by infrastructure providers. In addition, a function PlatformAware.setup() is offered for partial automatic feature detection and fills the Platform.toml file, but it works only for Linux environments.

Using dynamic multiple dispatch over platform types, platform assumptions of individual kernel methods can be added and modified independently, improving modularity and extensibility by addressing the platform-aware expression problem. We argue this is an improvement over ad-hoc platform-aware practices in Julia, where the programmer may be required to discover and use one or more packages to detect platform features, either interspersing detection code within the code of platform-aware functions or creating separate platform selector functions that must be modified to add or modify platform assumptions, as shown in [3].

Although it is realistic to assume that platform assumptions are local to each kernel function, PlatformAware.jl also supports specifying a set of platform types for all kernel methods, representing target platforms of the application. In such a global strategy, adding or modifying a platform applies to all corresponding methods for each kernel function, still not interfering with the code of existing kernel methods.

4 Final remarks

The expression problem helps measure the programming language's ability to deal with software modularity and extensibility issues. It emerged under the architectural independence premise in programming language design. However, the unprecedented performance requirements of AI and data analytics applications have boosted the interest in HPC techniques, such as parallel and heterogeneous computing, bringing platform-aware programming practices to general software development. This motivated us to restate the expression problem by breaking architectural independence.

PlatformAware.jl shows an approach to tackle the platform-aware expression problem, but it draws attention to the dynamic feature detection problem, leading to the implementation of a static solution. Dynamic feature detection is useful when the underlying physical platform may vary between executions, such as when using containers and virtual machines in cloud-based environments, but it is challenging to implement under current technology, as argued before.

We draw the programming language community's attention to the need for initiatives to improve platform-aware programming support. As the demand for computational capacity increases due to AI applications, motivating the emergence of special-purpose processors and accelerators, the optimal use of computational resources becomes a competitive differentiator in industrial and scientific applications.

References

[1] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proceedings of ACM Programming Languages* 2, OOPSLA, Article 120 (oct 2018), 23 pages.

[2] W. R. Cook. 1991. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–178.

[3] F. H. de Carvalho Junior, A. B. Dantas, J. M. Hoffiman, T. Carneiro, C. S. Sales, and P. A. S. Sales. 2023. Structured Platform-Aware Programming. In *XXIV Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD'2023)* (Porto Alegre, RS). SBC, Porto Alegre, Brazil, 301–312. https://github.com/PlatformAwareProgramming/PlatformAware.jl

[4] A. Grama, A. Gupta, J. Karypis, and V. Kumar. 2003. *Introduction to Parallel Computing*. Addison-Wesley. 256 pages.

[5] Matthias Z. and Martin O. 2005. Independently Extensible Solutions to the Expression Problem. In *12th International Workshop on Foundations of Object-Oriented Languages (FOOL'2005)*. ACM.

[6] H. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. 2013. *Top 500 Supercomputer sites*. http://www.top500.org

[7] F. Z. Nardelli, J. Belyakova, A. Pelenitsyn, B. Chung, J. Bezanson, and J. Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proceedings of the ACM Programming Languages* 2, Article 113 (oct 2018), 27 pages.

[8] B. C. d. S. Oliveira. 2014. Functional programming, object-oriented programming, and algebras!. In *10th ACM SIGPLAN Workshop on Generic Programming* (Gothenburg, Sweden). ACM, New York, USA, 1.

[9] N. C. Thompson and S. Spanuth. 2021. The decline of computers as a general purpose technology. *Communications of the ACM* 64, 3 (feb 2021), 64–72.

[10] M. Torgersen. 2004. The Expression Problem Revisited. In *ECOOP 2004 – Object-Oriented Programming*. Springer, Berlin, Heidelberg, 123–146.

[11] P. Wadler. 1998. *The Expression Problem*. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

[12] Y. Wang and B. C. d. S. Oliveira. 2016. The expression problem, trivially!. In *15th International Conference on Modularity* (Málaga, Spain). ACM, 37–41.

[13] W. Zhang, Y. Sun, and B. C. d. S. Oliveira. 2021. Compositional Programming. *ACM Transactions on Programming Language Systems* 43, 3 (sep 2021), 61 pages.