Future Programming Languages: What Can the Past, and Future Predictions, Say About Success of Potential Languages?

Athabasca University - Comp 601

February 19, 2017

Jason Bell 3078931

Professor: Dr. Huntrods

Abstract

The previous assignment for this course was a survey of the history of successful programming languages, and what they can teach us about implementing new ones. New programming languages should consider both the past and the future. Fortran is one of the oldest high level languages, yet is still extensively used for scientific computing. Why is this, and are these features worth considering in future languages? Successful languages sometimes revert to earlier, ostensibly more primitive forms. Can a future language make a similar reversion and be successful? What are potential challenges of the future that will impact language popularity?

Keywords: programming language, compiler, history, future

Introduction

For the first assignment of this course, I chose to survey the history of successful programming languages, and discuss what this can teach the creator of new programming languages. The hope was that this will reveal ways in which my own custom language, Pie, can be a successfully adopted one rather than an intellectual exercise.

This paper builds upon the previous survey, and discusses how to apply its lessons to future programming languages, such as Pie. It starts by discussing Fortran: what features does it have that make it desirable for scientific computing? It next discusses whether it can be worth-while for a future language to take a step backwards, as the C language did: to go “back to basics”. Are there features of modern programming languages, such as just-in-time compilation and garbage collection, that have seen over-adoption and could be removed or made optional? Lastly, what features are currently being discussed or desired for new programming languages?

The insights gained through this paper will be used in the design of a study under the supervision of Dr. Vive Kumar. This study will most likely take the form of a questionnaire compiled from ideas presented in this paper, and submitted to professionals and academics. Hopefully the resulting feedback will be useful, and will either shorten or lengthen the list of features of interest in a new language.

History

The historical survey of the previous assignment revealed that successful languages did one or more of three things. First, it may have introduced a very novel concept, such as code blocks in ALGOL. Second, it may have combined several pre-existing concepts into a well integrated whole for the first time, as was the case with Python. Third, and most interestingly, a language can be successful by taking a step back. The C language came about because of the needs of systems programmers, who desired a language that mapped more closely to machine code, allowed hardware access, and did not have a verbose English syntax.

Two programming languages are notable for still being in common use, despite being in the first generation of high level programming languages: COBOL and Fortran. While COBOL largely continues to exist due to large amounts of legacy code that is not worth the cost of replacing, Fortran is still actively used for scientific computing and supercomputers. Clearly, Fortran must provide unique advantages, for it to be favored over more modern languages.

The overall lesson of this survey was that a language must be more than just a clean and beautiful syntax. D is an example of a programming language that is just that: it has a clean C syntax, and tries to combine the power of C++ with ease of use [1]. Unfortunately, this description applies equally to languages such as Java and C#, which likely explains why D has not seen widespread adoption: this niche is already full.

Discussion

Fortran

Surveying users of Fortran reveals that there are many reasons why it is still used for scientific computing. The most common opinions were that the language is highly scalable, and that it provides a host of quality of life features for scientific computing. The Fortran compiler was the first to optimize code, and this principle is still core to the language. The precise details of how these optimizations are carried out is outside the scope of this paper, but are useful to discuss at a high level.

Fortran variables that are intended to be pointers must be explicitly defined as such. In C or C++, integers and pointers are interchangeable: a pointer is nothing more than an integer variable that happens to hold a memory address, and one can be cast to the other. In Fortran, a variable is either a pointer, or it is not. This allows the compiler to make assumptions about how it will be used and optimize accordingly. In the same vein, a variable can be declared as a target. This indicates that it can be pointed to by a pointer: therefore, a variable is capable of being pointed to, or not, allowing further optimization [2].

Expanding upon this, function arguments can not be aliases of each other: one can not pass the same variable to more than one argument, unless that variable is a target or pointer [3]. This allows further optimization: I hypothesize that it is a way of limiting the number of times that entire objects are copied when a function is invoked: pointers can be aliased as they are merely integer values that are cheap to copy, and targets can be aliased after the compiler substitutes it with a pointer that points to it. Further research will indicate if this is true, and if there are additional reasons.

Fortran has support for parallelization as a language feature: clearly a useful feature for a language running on a supercomputer with a huge number of parallel CPUs and memory. Fortran has a DO CONCURRENT statement, that executes its contents concurrently. Adding the CONCURRENT keyword to a DO statement is a guarantee to the compiler that the iterations of the DO loop can be treated as being independent [4]. Therefore, the compiler can execute the iterations concurrently rather than in a series. Similarly, the FORALL statement iterates over the contents of an array and executes statements on its elements concurrently [5]. These features make it easier to write, deploy, and debug concurrent code, important when one has finite time to use a supercomputer.

Co-Array Fortran adds further parallelization by extending Fortran with the Single-Program-Multiple-Data model. A single program is implemented, and is copied into multiple images. Each program image is assigned a subset of the data to be processed, and they are executed in parallel. Its use is very like Fortran without this extension: broadly, the primary difference is that applicable data variables must be declared with an additional syntax to indicate that it is distributed data [6].

Beyond these concurrency features, the most popularly cited features of Fortran are quality of life features as they apply to numeric computing. Arrays can have any user defined index range: the starting index does not have to be zero, which simplifies some computations [7]. Thanks to the language’s long legacy, there are many supporting libraries and established standards. A few examples include: BLAS, which describes linear algebra standards [8], EISPACK for calculation of eigenvalues [9], MINPACK for solving nonlinear problems [10], SOFA for astronomical calculations [11], and NAG for statistical calculations [12].

Back to C++

The survey carried out for the first assignment showed an interesting trend: another back-lash, at least in some contexts, against languages that are perceived as too high level. One example is IL2CPP, used by the Unity game engine to translate C# code to C++. While this is currently only supported for IOS, the goal is to use it for all platforms [13]: programs will be platform specific, but may be faster in some circumstances. The degree of how much faster is very debatable and situation dependent.

A future language that compiles to native machine code rather than being just-in-time compiled would have a simpler compiler and runtime: without the need for defining a robust intermediate language, and compilation through it. The down-side is that the resulting binaries would be platform specific rather than “build once, deploy anywhere”. I would suggest, given my admittedly limited professional programming experience, that this is a need that does not occur very often: I have never had to copy a library or executable straight from Windows to Linux. Instead, I have always used each platform’s package management tool to obtain them. Therefore, requiring a user to download platform specific builds seems like it would be of little to no inconvenience.

There may be other advantages of compiling to machine code, rather than just-in-time compilation. As a hobby, I often tinker with graphics and video game programming. Unless one is using a pre-made engine, this often requires use of native libraries: OpenGL, Direct3D, OpenAL, and so on. Doing so from a high-level language requires use of either third party glue libraries of sometimes questionable quality, or implementation of glue code of one’s own. Would it be possible to create a high-level programming language that allows one to “drill down” to that level? Is this a thing that some programmers would desire?

The current version of Pie compiles through C# as a bridging language, just as CFront, the first C++ compiler, used C as a bridging language [14]. This simplifies the first iterations of the compiler, as one does not need to worry about as many details: any compiler errors missed by the custom compiler will be caught by the compiler for the bridging language. This also allows the new compiler to take advantage of optimizations carried out by the bridging language compiler. While I had planned to remove this bridging step in the next version of Pie, perhaps this is unnecessary. Pie could become a very powerful language if it compiled through C++. I envision it as a high-level language layered on top of C++: being more convenient to use than C++, while retaining much of that language’s power and flexibility. In this sense, it would be a friendly but thin façade on top of C++, a façade through which one can “punch”.

Since Pie would already compile through C++, I see no technical reason why one could not have blocks of C++ code embedded inline in Pie code. While other languages, such as Cython, allow one to compile C++ files into the project [15], this would instead be C++ code mixed into Pie code. Rather than having to invoke an OpenGL function, one directly calls the C function, with no need for glue code.

Despite compiling through a static type language, Pie could still be dynamic type. A Pie “class” would simply be a C++ class, but derived from a base “Object” class that provides the infrastructure required for late binding: primarily a hash table in which to store class members, and a function that allows access to those members. The advantage of this would be making it convenient to offload performance critical code to C++, important since the hash table lookups required by dynamic type languages adds considerable overhead. One would write most the code in a convenient, friendly, dynamic type language layered on top of C++, and if profiling reveals performance bottlenecks, those sections could be converted to embedded C++ blocks.

Another feature I have considered for future versions of Pie is interopability with other high level languages. Prior to considering layering the language on top of C++, this seemed like it would be a very large technical challenge. However, most high level languages have libraries that allow their use from C++. Java can be invoked using Java Native Invocation (JNI) [16]. C# can be invoked by embedding the Mono runtime into the application [17]. Python provides extremely robust support for invocation from C++ [18]. The lesson is that this goal will become much easier if Pie compiles through C++. Plugins for each language would wrap that language's respective C++ binding library, and return dynamic type objects. Rather than performing a hash table look-up when one of the object's members is invoked, the object would instead use the C++ binding library to invoke that member in the target language.

Clearly, compiling through C++ will provide many advantages and could be very appealing, if it's done in a clean and robust way. There are sure to be challenges. For example, I would prefer that the Pie compiler not care about order of declaration, as is the case in C# and Java. However, order of declaration matters in C++, so a solution to this needs to be investigated. I do not expect this to be an insurmountable challenge: I speculate that the worst case is that the Pie compiler will need to insert large numbers of forward declarations while compiling into C++.

The Future

For this section, I surveyed the internet for discussion about what features are of interest to future languages, and what challenges the future will bring. There are a huge number of young and upcoming languages, and discussion of all of them is impossible. I will forego discussion of certain domain-specific languages, such as R, as I envision Pie as general-purpose.

A common theme is that without a radical advance in technology, we are nearing the limits of how much processing an individual CPU can do. This, combined with the explosion in cloud computing, means that excellent support for distributed and parallel processing is of paramount importance. This is also why Fortran is still used for scientific computing on supercomputers: a feature that has resulted in a language created in the 1950s still being used, must be an important feature indeed. Clearly, any future language must provide excellent support for distributed and parallel processing. I would go so far as to suggest that this is non-negotiable, and a language that does not is doomed to failure.

In the same vein, support for distributed data seems like a requirement for any new language. Again, this is a feature available in Fortran that contributes to it's popularity. With the explosion in cloud computing, it is important that a new language allows declaration of variables as distributed data.

Elm is a language with several interesting features. Most are highly domain-specific: the language is clearly geared towards web technologies. I will not discuss these features, as I envision Pie as a general-purpose language. The one feature noteworthy in the context of Pie is Elm's "enforced semantic versioning". Packages in Elm's package manager are strictly required to follow well defined semantic standards, with the goal of eliminating dependency issues and conflicts often found in other package managers [19]. This feature is of note to me, as it was frustration with conflicting dependencies in a Java Maven project that inspired me to create my own compiler. Clearly, an excellent package manager is important for any new language: especially if Pie compiles through C++ and has platform-specific binaries.

In the theme of “back to C++”, the Rust language has no garbage collector. Instead, it employs RAII (resource acquisition is initialization) and C++ smart pointers to ensure memory safety, without the overhead of garbage collection [20]. This interests me both for the lessened overhead, and for the simpler runtime. Perhaps it would be desirable to give options: a choice between no memory management, light-weight management with smart pointers, or garbage collection.

Many new programming languages are designed around, or support, the functional programming paradigm. The primary advantage of functional programming, aside from personal preference, is the elimination of “side effects”. The output of a function is entirely dependent upon it’s input, making it easier to isolate problems. Contrast with the object-orientated paradigm, where objects have states. If implemented poorly, those states can potentially be altered from elsewhere in the code, at any time. Examples of recent general-purpose functional programming languages are F# [21], Scala [22], and Elixer [23].

I will end this section with some speculation of my own. Clearly, going forward, projects are only going to get bigger and more complex. What can a language, compiler, or IDE do to assist with this challenge? Pie already supports unit testing and test coverage analysis as language features, and the efficacy of test driven development was more than proven to me. This is a feature that the language must retain.

Are there additional features that can help with complex and large projects? Perhaps design patterns as a language feature? Admittedly, I am not an avid user of design patterns: my code tends to consist of loosely coupled modules. Therefore, I have no wisdom of my own as to whether this is something that would be desirable, beneficial, or even practical. The answer to that will require more research, and surveying professional programmers. I can speculate, however, that this may be of interest to project managers. Perhaps the tools can be designed in such a way that the project manager can set certain rules and standards. A member of the team is not able to submit code unless it adheres to a certain design pattern, or uses a specific paradigm, or has some minimum level of test coverage. These features should be entirely optional for the general user, but I speculate that they would be of great interest to project managers.

Conclusion

The most powerful message of this discussion is that support for distributed and parallel programming is of top priority: preferably as language features rather than third party tools. This includes support for both distributed computing and data as largely transparent language features. The programmer should not need to know what degree of parallelization or distribution is occurring: it should be possible to implement a program one way, and have it run the same on a single CPU, multiple CPUs on the same computer, large numbers of CPUs on a supercomputer, or distributed in a cloud. Distribution of instructions and data should occur with as little intervention by the user as possible. It is these qualities that explain the continued success of Fortran, despite competition from more modern languages.

Further conclusions are difficult to make without more experience or feedback. This feedback will be gained through a questionnaire provided to academics and professionals, a draft example of which follows. Of the following features, which, if any, are of interest for a new language? Why or why not?

- Ahead-of-time rather than just-in-time compilation.
- Dynamic type.

- Design patterns as language features.
- Unit testing and test coverage analysis as language features/
- Embedded C++ code inline with the new language, allowing more convenient access to native libraries and offloading of high performance code to C++.
- Support for a package manager with very strict rules to prevent dependency conflicts.
- Interopability with other high level languages though plugins, if those languages are invokable from C++.
- Configurable memory management: either none, light weight with smart pointers, or garbage collection.
- Support for multiple programming paradigms: C++ already supports, at the minimum, procedural, object-oriented, and functional programming. A language built on top of it should be able to as well.
- The ability for a project manager to configure compiler settings for the entire team. Require a minimum level of test coverage, use of design patterns, and specific paradigms.

Citations

[1] D Language Foundation. (2017). *D Programming Languag*e [Online]. Available: https://dlang.org/

[2] Queen's University of Belfast. (2017). *Pointer Variables* [Online]. Available: http://www.pcc.qub.ac.uk/tec/courses/f90/stu-notes/F90_notesMIF_12.html

[3] Wikipedia. (2017). *Pointer Aliasing* [Online]. Available: https://en.wikipedia.org/wiki/Pointer_aliasing

[4] International Business Machines (IBM). (2017). *DO CONCURRENT construct (Fortran 2008)* [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSGH4D_15.1.3/com.ibm.xlf1513.aix.doc/language_ref/do_concurrent_construct.html

[5] University of Liverpool. (1997). *Forall Examples [Online]*. Available: http://www.fortran.gantep.edu.tr/local/HPFCourse/HTMLHPFCourseNotesnode235.html

[6] Numrich, R. W., & Reid, J. (1998, August). Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum* (Vol. 17, No. 2, pp. 1-31). ACM.

[7] W. Brainerd, C. Goldberg, and J. Adams. (1994). *Programmers Guide to Fortran 90, 3nd (sic) Edition. Chapter 4 Arrays* [Online]. Available: http://www.fortran.com/fortran_storenew/Html/Info/books/gd3_c04_1.html

[8] Netlib Repository. (2016). *BLAS (Basic Linear Algebra Subprograms)* [Online]. Available: http://www.netlib.org/blas/

[9] Netlib Repository. (2016). *EISPACK* [Online]. Available: http://www.netlib.org/eispack/

[10] Netlib Repository. (2016). *MINPACK* [Online]. Available: http://www.netlib.org/minpack/

[11] International Astronomical Union. (2016). *Standards of Fundamental Astronomy* [Online]. Available: http://www.iausofa.org/

[12] Numerical Algorithms Group (NAG). (2017). *The Nag Fortran Library* [Online]. Available: https://www.nag.com/nag-fortran-library

[13] Unity Game Engine. (2017). *An Introduction to IL2CPP Internals* [Online]. Available: https://blogs.unity3d.com/2015/05/06/an-introduction-to-ilcpp-internals/

[14] Bjarne Stroustrup. (2016). *Bjarne Stroustrup's FAQ* [Online]. Available: http://www.stroustrup.com/bs_faq.html

[15] Cython. (2017). *Using C++ in Cython* [Online]. Available: http://cython.readthedocs.io/en/latest/src/userguide/wrapping_CPlusPlus.html

[16] Code Project. (2015). *Calling Java from C++ with JNI* [Online]. Available: https://www.codeproject.com/Articles/993067/Calling-Java-from-Cplusplus-with-JNI

[17] Mono Project. (2017). *Embedding Mono* [Online]. Available: http://www.mono-project.com/docs/advanced/embedding/

[18] Python Software Foundation. (2017). *Embedding Python in Another Application* [Online]. Available: https://docs.python.org/2/extending/embedding.html

[19] Evan Czaplicki. (2017). *Elm* [Online]. Available: http://elm-lang.org/

[20] Rust. (2017). *Frequently Asked Questions* [Online]. Available: https://www.rust-lang.org/en-US/faq.html

[21] F# Software Foundation. (2017). *F#* [Online]. Available: http://fsharp.org/

[22] École Polytechnique Fédérale de Lausann. (2017). *Scala* [Online]. Available: https://www.scala-lang.org/

[23] Plataformatec. (2017). *Elixir* [Online]. Available: http://elixir-lang.org/