Preventing Babel: Rectifying the Trend of Programming Language Divergence

Alon Amid
University of California, Berkeley
alonamid@eecs.berkeley.edu

Borivoje Nikolic
University of California, Berkeley
bora@eecs.berkeley.edu

Abstract

Throughout the history of computers, there has been a proliferation of new programming languages. Programmers have a variety of language choices for application or system development. Many languages have differing constructs, syntax rules, and development environments, which impede the productivity of programmers using a diverse language set. With the renewed interest in Domain Specific Languages (DSL), we argue that further emphasis should be put on common core constructs and syntax during the design and evaluation of a new language. We believe common base languages will improve programmers' productivity, and allow the development of a strong supporting ecosystem of programming systems.

1 Introduction

Programming languages are the programmer's main tool for interacting with a computer, hence, they should be a skill at which programmers are highly proficient. However, the fast evolution, and increasing variety and diversity of programming languages makes it increasingly harder to reach a high level of proficiency for many programmers. De-facto standardization and common cores have historically proven to be successful at supporting highly proficients communities which drive powerful ecosystems in other parts of the computing stack (ISA, OS). On the other hand, the diversity of programming languages allows for domain-specific applications and systems to be expressed more efficiently and accurately. We must ask, how can we balance the benefits of the diverse set of domain oriented programming languages with the power of strong core ecosystems.

2 Programmers and Languages

In recent years "Google Programming" or "Stack Overflow programming" has become a dominant form of coding. This form of "programming" consists of performing a Google search, or checking for answers in the Stack Overflow website archive in order to write basic program blocks. These are indeed great tools, which allow for improved knowledge management and quick debugging by experience sharing. However, given that programming languages are the basic tool used to express specification-based instructions to a computer, we argue that a minimum proficiency should be acquired for efficient use of a programming language.

Programming efficiency is diminished when programmers stop working in order to search for the correct form of simple operations such as variable assignments, conditional statements, or loop declarations. As an example, a basic question such as "how to write an if-else statement in Python" has approximately 200,000 views on Stack Overflow at the time of writing [6]. While this is not a precise and accurate measure of the popularity of this question (since the answer may also be found in simple tutorials and Google searches), it does provide a sense to the extent of programmers' knowledge as to the basic behavior of a popular programming language. The lack of thorough knowledge of a programming language's basic behavior may lead to simple compilation errors, as well as more concerning logical bugs. These types of bugs are prominent when switching between 0-based and 1-based indexed languages, or languages that differ in calling by reference/value/name. Furthermore, when programmers are not highly proficient in a language, they will not be able to use its unique and powerful traits which are many times the justification for the diversity of languages.

A principle reason for this lack of high-level proficiency is due to the difference in the implementation and representation of basic constructs between languages. Gaining proficiency in a programming language requires language learning and acquisition skills. Unlike other professional tools or libraries, a language consists of many components such as syntax, semantics, morphology and vocabulary, just to name a few. While programming languages are usually less verbose than human natural languages (especially in vocabulary), we argue that similar proficiency levels will lead to more efficient use of programming languages. We define efficient use as requiring less time to write a given functionality, and using the least number of instructions to describe that functionality.

According to [4], about 54% of Europeans speak at least 2 languages, while only 10% speak at least 4 languages. [5] found that a majority of programmers identify as proficient in approximately 4-6 programming languages, and require between 1 month to 1 year to learn each language. An undergraduate student in a computer science or electrical engineering program will likely use at least 5 programming

PLATEAU'17 Workshop on Evaluation and Usability of Programming Languages and Tools, October 23, 2017, Vancouver, CA

languages during their studies, according to course curriculum [11]. Since learning a programming language requires many of the same skills as acquiring a natural language, we must wonder why do we allow and require ourselves to learn so many more programming languages.

3 Domain Specific Languages

Domain Specific Languages (DSLs) have existed for many years as independent environments. Some examples include SQL for relational databases and Verilog/VHDL for hardware description. These languages have proven to develop unique ecosystems around them, but were bound to large enough industries which could support these types of independent silo environments. Knowledge of these languages was part of a relevant professional's toolkit and basic training, such as data analysis or hardware engineering.

DSLs have experienced a resurgence in popularity in recent years, in order to provide better abstractions for programmers to use in specific contexts. Some examples include P4 [2] in the networking community, Halide [7] for image processing, and Chisel for hardware description [1]. We predict this trend will further expand with the evolution of Domain Specific Accelerators in the hardware community, which further motivate unique programming abstractions.

While we agree that there exists a need for these languages, and they indeed provide powerful abstractions for their respective purposes, we also maintain that in many cases these abstractions do not require re-inventing the wheel and generating complete new languages. We believe that new abstractions can evolve to extend existing core languages, just as natural languages evolve gradually with the introduction of new concepts (through vocabularies, semantics, and contractions). Examples of DSLs extending core languages include Halide which is embedded in C++, and Chisel which is embedded in Scala (the definition of Scala as a core language can be argued). On the other hand, other DSLs such a P4 provide their own base language due to the argument that they may target different hardware platforms.

4 Discussion and Proposals

The language divergence problem is highly visible in dynamic and scripting languages. Languages such as Python, R, Matlab and Perl are used interchangeably for similar purposes of data analysis and manipulation. They are all high level, dynamic, interpreted languages which have a similar core functionality and significant reliance on their library and module communities. These languages are many times associated with user communities due to historical reasons rather than domain-specific characteristics. Arguably, much of the domain-specific functionality of these languages is derived from their library communities [5]. While these languages may be used interchangeability for many purposes, they have different syntax, constructs, and supporting ecosystems. As an example, when a Matlab-trained engineer works jointly with Python-trained scientists, one of them must learn a new syntax, new type constructs, new module libraries, and invest significant debugging time in order to perform simple actions that both scripting languages perform equivalently. This type of knowledge transfer and vocabulary translation is especially damaging, as it is not domain-related, and it adds another level of burden to the task of domain-specific knowledge transfer.

We are not the first to question the reasons behind the large number of programming languages [9], nor are we the first to speculate regarding the impact of the historical evolution rather than design [8]. However, we argue that further actions should be taken in order to prevent further exponential growth of this phenomenon due to the addition of the specialization dimension. While history has showed that programming languages which do not meet mainstream conventions end up disappearing overtime through the laws of "natural selection" and "free market economy", this process costs many resources and a long time. With the rise in popularity of DSLs, the programming community is coming to an important junction: allowing the historical evolution to continue encouraging even further divergence, or setting compatibility as a priority for DSLs. We propose using syntax and basic semantics as a tool to unify basic language constructs. Compatibility and syntax standardization should take a stronger role when designing and reviewing new languages, as important as the role of the programming paradigm and abstraction. Previous studies [10] examined the effects of syntax on novice and experienced programmers. While the analysis focused on novice programmers, the results show that programmers with previous programming experience have a different understanding of "intuitive syntax" compared to novice programmers. This is further supported by natural language learning research, which distinguishes between primary language acquisition and secondary language acquisition [3]. Since in most cases DSLs are not the first programming language learned by developers, design considerations for DSLs should be different than those of a general purpose language, and support the case for common syntax and semantics cores based on historically popular general purpose languages. DSLs should be designed as extensions of existing core languages (sometimes referred to as "Internal/embedded" DSLs), rather than new languages with their own syntax and semantics. While it is not possible to control which languages gain popularity, guidelines and methodologies for the development of new languages may assist in eventual convergence. We hope that common primitives and syntax families will consolidate into general cores that can be supported by common tools and infrastructure for a variety of extended languages.

As an example, the core structure of the C language syntax has proven useful for many compiled languages due to its expressive native primitives. Hence, DSLs with performance

oriented goals should extend and maintain compatibility with a C-family language. Analogously, usability or productivity oriented DSLs, DSLs that utilize meta-programming, or DSLs which do not generate machine-target code, could use Python as a common core, since it has emerged as the language of choice for general purpose scripting. Usability and performance generate trade-offs that can be proved to conflict with each other, and therefore justify different core families. Divergence should be justified only by formally proving inherent conflicts with a core language. While the constraints of extending existing programming languages add challenges compared to designing a language from scratch, these challenges should be embraced rather than imposed later on to the users.

5 Conclusion

The title of this paper is derived from the biblical tale of the tower of babel, in which the different languages diverged and prevented humanity from reaching the next great achievement. While DSL popularity is increasing, we argue for further emphasis on language compatibility, and prevention of language divergence using common base syntax and constructs. Language extension and compatibility assists in avoiding the long process of "natural selection" of programming languages. These arguments build upon historical precedence of de-facto standards, and analogies between programming languages and natural languages. DSLs are a powerful tool, but they should be structured to support common language cores to drive the development of strong ecosystems. These language cores will benefit both programmers and programming-system designers who will be able to build upon powerful ecosystems with highly proficient users.

6 Acknowledgment

The authors are partially supported by DARPA Award Number HR0011-12-2-0016 and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HPE, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors. The authors would like to thank Jonathan Reagen-Kelly for the informative and thoughtful discussions regarding the topics mentioned in the paper.

References

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1216–1225.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[3] Catherine J Doughty and Michael H Long. 2008. *The handbook of second language acquisition*. Vol. 27. John Wiley & Sons.

[4] Special Eurobarometer. 2012. Europeans and their Languages. *European Commission* (2012).

[5] Leo A Meyerovich and Ariel S Rabkin. 2013. Empirical analysis of programming language adoption. *ACM SIGPLAN Notices* 48, 10 (2013), 1–18.

[6] Stack Overflow. 2010. What is the correct syntax for 'else if'? (2010). https://stackoverflow.com/questions/2395160/what-is-the-correct-syntax-for-else-if

[7] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[8] Matt Sherman. 2015. Why are there so many programming languages? (2015). https://stackoverflow.blog/2015/07/29/why-are-there-so-many-programming-languages/

[9] Andreas Stefik and Stefan Hanenberg. 2014. The programming language wars: Questions and responsibilities for the programming language community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 283–299.

[10] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 19.

[11] EECS Department University of California, Berkeley. 2017. EECS Major Requirements. (2017). https://eecs.berkeley.edu/resources/undergrads/eecs/degree-reqs