Towards a Formalism for Computing Platforms

Daniel Campos do Nascimento

February 20, 2024

Abstract

Software development seems to have evolved enormously since the advent of the stored-program computer, but ideas currently in use have precedents even in the 1960s. Meanwhile, the lambda calculus is essentially unchanged since its introduction by Alonzo Church in 1936, and even though it was created as part of research into the foundation of mathematics, its link to computing was recognized early on. This suggests that the groundwork for a formal description of commonly known software development phenomena, currently incarnated in computing platform feature sets, is already possible; yet no formalism is cited anywhere, if it even already exists. This paper will do a first attempt at such a description in three steps. First, the theoretical elements needed for it will be presented. Then the phenomena of software development under study will be determined via a historical analysis. Finally the formalism will be described and discussed.

1 Theory

1.1 Lambda calculus

Though it is well-known, a brief definition of the lambda calculus will be given nevertheless. Define the notion "lambda term" such that

1. x is a lambda term called a *variable*.
2. If M is a lambda term, $\lambda x.\ M$ is a lambda term called an *abstraction*.
3. If M and N are lambda terms, MN is a lambda term called an *application*.

These rules describe how to write programs in the calculus. Next define $M[x := N]$, substitution, read as "M where N substitutes x", to be M with each occurrence of x replaced by N. Then define $FV(M)$ such that

1. $FV(x) = \{x\}$.
2. $FV(\lambda x.\ M) = FV(M) - \{x\}$.
3. $FV(MN) = FV(M) \cup FV(N)$.

Finally,

1. β-reduction: $(\lambda x.\ L)N \to_\beta L[x := N]$
2. α-conversion: $\lambda x.\ L \to_\alpha \lambda y.\ L[x := y]$
3. η-reduction: if $x \notin FV(f)$, $\lambda x.\ fx = f$

These six rules are by no means necessary for all computations, but they constitute the most complete description agreed upon by the community.

1.2 Partial evaluation

One will observe that abstractions bind only one variable, applications apply only one term to one value, and substitution is defined only for one variable at a time. Thus all abstractions are already *curried*, or in a normal form such that more familiar multivariable functions are represented by a lambda abstraction binding one variable of the function and returning a function of the remaining variables to be bound, and thus all applications are partial applications. Therefore, if application is considered left-associative,

$$LMN = (LM)N$$

therefore by η-reduction,

3. If $y \notin FV(fx), \lambda y.\ fxy = fx$

To illustrate in more detail, if abstraction is considered right-associative,

$$\lambda x.\ \lambda y.\ M = \lambda x.\ (\lambda y.\ M)$$

therefore

$$\begin{aligned}(\lambda x.\ \lambda y.\ Lxy)MN &\to_\beta (\lambda y.\ Lxy)[x := M]N \\ &= (\lambda y.\ LMy)N \\ &\to_\beta (LMy)[y := N] \\ &= LMN\end{aligned}$$

Alternatively,

$$\begin{aligned}LMN &=_\eta (\lambda x.\ Lx)MN, x \notin FV(L) \\ &=_\eta (\lambda x.\ \lambda y.\ Lxy)MN, y \notin FV(Lx)\end{aligned}$$

This will be critical to the next concept: partial evaluation.

It is important to not confuse partial evaluation with partial application; partial application is merely the **result** of applying a curried function to a single argument, and partial evaluation is an implementation of partial application. With currying, partial evaluation becomes almost trivial to describe: the partial evaluator is just

$$\lambda f.\ \lambda x.\ fx$$

Note that all variables are bound, so the partial evaluator is also a combinator; it will herein be called E.

1.3 Types

It is important to note that one reason for this description's triviality is a lack of types; indeed, that fx itself might be an abstraction appears nowhere. To make this more explicit, the simply typed lambda calculus will be employed. In the simply typed lambda calculus, define the notion of "types" as follows:

1. Σ is a type called a *base type*.
2. if σ and τ are types, $\sigma \to \tau$ is a type called a *function type*.

Types then follow the following rules:

1. $$\frac{}{\Gamma \vdash c{:}\Sigma}$$
2. $$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma}$$
3. $$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)}$$
4. $$\frac{\Gamma \vdash e{:}\sigma \to \tau, \Gamma \vdash f{:}\sigma}{\Gamma \vdash ef{:}\tau}$$

Additionally, lambda calculus rule 2 is changed to:

2. If M is a lambda term, $\lambda x{:}\sigma.\ M$ is a lambda term called an *abstraction*.

And η- and β-reductions are changed to:

1. β-reduction: if $\Gamma, x{:}\sigma \vdash L{:}\tau$ and $\Gamma \vdash N{:}\sigma$, $(\lambda x{:}\sigma.\ L)N \to_\beta L[x := N]$
3. η-reduction: if $x \notin FV(f)$ and $\Gamma \vdash f{:}\sigma \to \tau$, $\lambda x{:}\sigma.\ fx = f$

With this, if $\to$ is considered right-associative, then

$$\sigma \to \tau \to \upsilon = \sigma \to (\tau \to \upsilon)$$

therefore E becomes

$$\lambda f{:}\sigma \to \tau \to \upsilon.\ \lambda x{:}\sigma.\ fx$$

for all σ, τ and υ.

The previous sentence suggests a greater universality is possible, namely parametric polymorphism. Nevertheless, parametric polymorphism will not be employed here for the reason that its feature set is out of this article's scope.

1.4 Higher-order abstract syntax

In practice, the concept of a variable in practical programming languages differs slightly from the notion of a variable in lambda calculus. Fortunately, a construct exists which captures this exact relationship quite nicely: higher-order abstract syntax (HOAS).

Note that the HOAS employed here is as employed in the context of logical frameworks. Thus, the HOAS of

let $x = f$ in x^2

is

let $f\ \lambda x.\ x^2$

Even though other HOAS exist and are equally valid, the above HOAS was chosen for its slightly greater flexibility and applicability to the article's subject.

2 Phenomena

2.1 History

The genesis of modern software development began with the first stored program computers. Programmers wrote in machine code only for a brief time; the first assembler was created for the EDSAC in 1949. From there many concepts came relatively quickly: subroutines were created to factor out common code and reduce code size, and Alan Turing had anticipated the modern return address stack in 1945 in a monograph for the ACE. The first modern software tool, however, was the A-0 system written by Grace Hopper for the UNIVAC I from 1951 to 1952. It worked more as a linker, a program being a list of subroutines with the arguments to be calculated; nevertheless, it was arguably the first machine-unrelated language ever implemented. Concurrently, Corrado Böhm described the first self-hosting compiler in his dissertation, submitted in 1951 and published in 1954[1]. FORTRAN appeared in 1957 and pioneered many concepts of compiler construction, principally optimization. LISP appeared in 1958 and pioneered the interpreter, as well as homoiconicity, the representation of programs in a language as instances of data structures in said language. COBOL appeared in 1959 and championed portability, then ALGOL 60 introduced many features considered characteristic of modern programming language, from formal grammars to block scope and recursion.

From that point onwards, many languages were compared to ALGOL 60 in behavior, and it so influenced programming languages that it spawned two major linguistic families: Pascal of 1970, and C of 1972. The Pascal branch was mostly used in niche applications, but the C family, bolstered by its usage in Unix, went on to establish a pattern for language implementation that went unchallenged until well into the 2000s.

Concurrently, though languages soon acquired a recognisable form, their usage was still conforming to the platforms, and more specifically the hardware, of the time: data generally was stored on punch cards, magnetic tape or, rarely, magnetic disk. Packages such as Input-Output Control System or IOCS provided the only (mildly) generic interface for programs to control the storage hardware so as to access data. This did not change until Multics arrived in 1964: the concept of single-level store, a file system and dynamic linking provided a programming environment unlike any seen before. Unix was inspired by Multics, retaining only the file system and the shell, but would add dynamic linking again with the arrival of the X window system. Multics thus first established the feature sets expected of any modern platform - and also established an accompanying software development pattern that persists to this day.

The first language which arguably challenged both patterns, surprisingly, was TEX, with the Comprehensive TEX Archive Network or CTAN; it would then inspire the Comprehensive Perl Archive Network, or CPAN, in 1993. These archive networks were the first challenges to the pattern because they brought package distribution and deployment, traditionally a concern of the platform, out of the operating systems and into the languages themselves; prior to this, the only programs distributed were either source code or binaries. Concurrently, Java introduced the concept of packages which, though implemented in the host filesystems, remained agnostic of them, and thus provided theoretical a way for the language implementation to handle all dependencies on its own. Both examples are important because they turned these languages from mere languages into software development platforms, illustrating the importance of the latter, through their success, to modern software development. Additionally, their interpreted or JIT-compiled nature cemented the role of the interpreter as the conveyor of the runtime, and of the runtime as incarnation of the platform, thus heralding the current trend of language-specific package archives, ecosystems and distribution networks.

It is thus clear that most of the progression in software development technologies is intrinsically linked to the capability of more powerful computers to support more powerful software development platforms. Languages have indeed evolved, but insofar as they are all Turing complete (in theory), they are all equivalent in expressive power. Different platforms, on the other hand, have wildly different characteristics depending on their feature sets - for example, a system without dynamic linking cannot achieve dynamic loading, at least not without self-modifying code.

2.2 Overview

Though there are many related activities, at its core software development is about specifying the behavior of a machine through code, an effective procedure for the calculation of a function. The procedure is described in code, and the particular input is supplied by the user as data. This development normally proceeds in three major phases:

1. specification;
2. translation;

3. testing.

Of these, translation is often the most difficult, just as it is between natural languages. Originally, humans translated ideas and specifications directly to machine code; as the specifications became more complex, two major trends happened:

1. humans took to expressing them in higher-level languages which could be automatically translated by the computer instead of by the human. Among other things, these languages often provide several common facilities, including:
   - control flow, either by virtue of being imperative/impure or through provided facilities (e.g. Haskell's do monad);
   - scope, through blocks organizing the imperative code;
   - modularity either directly (e.g. Java or Modula packages) or indirectly (e.g. C extern symbols);
2. platforms developed to factor out most common abstractions developed previously individually and ad-hoc. Many features developed, but among those chosen for this model are:
   - persistence;
   - concurrency; and
   - dynamic linking.

These features therefore constitute the phenomena which the formalism must account for. Moreover, it becomes clear that the language itself, the actual means by which to specify the code, is only one half of the observed evolution of software development; this remark will be expanded upon later. Therefore, while language theory is very well-developed, platforms are still very much ad-hoc and unformalized.

3 Formalism

3.1 Description

Every program obviously corresponds to a lambda term, but the formalism more specifically specifies *how* this lambda term corresponds to the program and, more precisely, how evaluation of this term corresponds (sometimes *directly*) to software development phenomena. It depends on exactly two techniques:

1. the representation of **all** references as free variables (higher-order abstract syntax); and
2. partial evaluation of the resulting term.

In essence:

- in a lambda abstraction, the bound variable is a symbol, and the body is a tree of references to variables, bound or unbound. A symbol table is therefore a collection of such bound variables, which may also be called *external symbols*, and the references are the free variables of the abstraction body;
- in an application, a value substitutes a variable in the body of an abstraction. Linking is thus a special case of this substitution, concerned more with function values rather than all values. In particular, with partial evaluation:
  - static linking corresponds to linking abstractions before linking arguments;
  - dynamic linking corresponds to linking arguments before linking abstractions;

3.2 Implications

From this description, several observations follow:

- The vast majority of all programming languages are about first-order calculations, specifying merely functions, whereas most module systems, whether defined with languages or not, are about second-order calculations, specifying function transformations or manipulations;
- compilation corresponds to translating statically linked abstractions;

3.3 Critique

The simplicity of the formalism might also raise questions regarding its time of description; the simple answer is history, and it is not meet for an article in computing to go into history, but given that a brief history of programming languages was already given, it will not do better to refrain from doing so. There are two major explanations as to the relative tardiness of such an analysis of platforms:

1. the most successful platforms were all developed by commercial enterprises, which were not even necessarily versed in computing theory, let alone abstraction. The concept of multiple implementations of a single architecture was pioneered by the IBM 7090, a transistor implementation of the 709, and fully realized in the System/360; the latter was first released in 1965, more than ten years after the first commercial computers, whereas LISP had been in existence for almost seven years.
2. most computer architectures were, and still are, based on automata, which, though easier to understand in isolation, do not encourage modular thinking.

4 Conclusion

It has thus been suggested that the lambda calculus is an adequate description and thus formalism for a fully featured software development platform. Software development was shown to have a historical origin that was rooted in an early division between language and platform; while language theory became well-established, platforms still developed in an ad-hoc and unformalized fashion. The resulting platforms had widely varying feature sets and equally widely varying behaviors which frustrated portability; at the same time, the advent of modules and package networks provided means for languages to take matters of software distribution and even dependency checking, a traditional concern of platforms, into their own hands. This article has suggested the suitability of the original lambda calculus as not just a programming language, but a programming environment and development platform itself, by establishing correspondances between platform features and phenomena of the lambda calculus' execution model. This article thus describes a first step towards a formalism for computing platforms, and furthermore suggests that such a formalism might be well within reach of current technologies, enabling applications hitherto unforeseen and new software systems well adapted to the problems of today and tomorrow.

References

[1] C. Böhm, "Digital Computers: On encoding logical-mathematical formulas using the machine itself during program conception," 1954.