The future of programming languages

Patrice Ossona de Mendez, Hubert Tonneau

Software 2.0

Number 100, April 2003, Pages 56–58 (polish translation)

THE FUTURE OF PROGRAMMING LANGUAGES

PATRICE OSSONA DE MENDEZ AND HUBERT TONNEAU

Abstract. The orientation of programming languages seems to follow two main directions: The first one corresponds to a seek of major companies to get more and more power over end users, while limiting their own responsibility in a kind of "democracy" under trusteeship. The other one is a quest for independence of the users expressed, for instance, in the free software movement. While the first tendency leads to a severe limitation of programmers' possibilities and to a uniformization of "personal" software restricted to "macro-softwares", the needs of the second one for coherency might lead to the introduction of another kind of operating system responsible for a language-independent compilation, which we shall call a *language operating system*, allowing a new level of integration: *language-level integration*.

1. Introduction

The first question we may ask in order to determine a possible future for programming languages is: "what is a programming language good for?"

It is no doubt that several issues are concerned with the programming of a computer application: at first, we want to be able to tell a computer what it should do, how it should do it, when it should do it and how it should interact with its environment. Each of these needs some "way to tell it", that is: some language.

Further characteristics will depend on our point of view and on our actual perspectives.

As well as the introduction of reading and writing has first led to the creation of an oligarchy somehow owning the techniques and using the growing needs of the society they belonged to to increase their power and to establish their supremacy (think about the power of the scribes in the antic Egypt), the introduction of programming techniques has first led to the a pyramidal organization of the knowledge and of the actual development power. Its is no doubt that the major companies that emerged in this context tend to keep their monopolistic position. Hence, they will be concerned by the preservation of their advantages, a limitation of their responsibilities and an increase of the dependence of their customers. These goals have to be kept in mind, when imagining the future of programming they may propose.

Nevertheless, coming back to our comparison, an irresistible movement toward a mass alphabetization emerged worldwide and, as the alphabetization became quasi universal, access to the knowledge and to the publication induced new democratic structures. Similarly, the access to programming cannot stay a reserved territory for some upper classes or software trusts. In that sense, the "free software" movement, although initiated by academics, has found a wide audience in the public, as an opportunity of democratization of the software development. From this later point

Patrice Ossona de Mendez is a researcher in discrete mathematics, co-developer of PIGALE (http://pigale.sourceforge.net) and member of Pliant project's core team (http://pliant.cx).

Hubert Tonneau is the development director of Heliogroup company, leader of Pliant project and creator of Pliant language.

of view, the perspectives are quite different: instead of a uniformization and a restriction of the programmer possibilities, one seeks the actual possibility to anyone to put its own brick on the edifice, to get the power of modifying the applications he uses and to progressively extend his programming skills to all the aspects of software development. This raises the problem of a global consistency in application interactions and the ability to reach each possible abstraction level.

Each of these points of view lead to a different probable evolution of programming techniques and, specifically, to divergent conceptions of the future of programming languages.

2. The oligarchic attempt

There is no doubt that the deep tendencies of the last years have been induced by majors software companies, which still possess the main part of the communication capabilities and a quasi trust on software markets. What are these tendencies?

- The introduction of pay-per-use applications, software rental, limited lifetime DVD, etc.
- The restriction of the programming access through proprietary intermediate code, which is interpreted or run into a virtual machine.

What are their goals?

First, users won't own anything anymore. Hence, no private copy will be allowed anymore, the responsibility of the software company will be reduced to each single use, the continuation of users' work will depend on its acceptance for a periodic update of a software he knows nothing about, using a protocol which is not clear. Thus, no responsibility on one side, and a total dependence on the other side.

Then, proprietary intermediate code will become the only possible access to the operating system and to the window manager. Any programming language will be heavily dependent on the actual possibilities of the intermediate code and of its libraries. After a while will only survive two kinds of programming languages: proprietary ones allowing to write some applications of limited complexity and efficiency, and more or less public ones appearing as kind of unified macro languages to pilot proprietary applications through "standard" libraries that you could not write within this framework. Here, object oriented programming is not seen as a way to properly distinguish parts of a software, but rather of a way of limiting the possible "intrusion" of a programmer in proprietary data structures.

Although all of this would look pretty and accessible (simplified, or even by-click programming within the world of some company), it would be much like a "democracy" under trusteeship, where the trustee would handle your resources as if they were its own, and where citizens would be more or less constrained to always reelect the some leaders, that would be less and less responsible for their acts. But, it is a common temptation for citizens to avoid any kind of personal responsibility, even by a loss of freedom...

Nevertheless, it doesn't mean that the idea of introducing intermediate code is bad *a priori*, as long as it is seen as an alternative choice for applications needing neither efficiency nor low level features. But recall that the introduction of an intermediate level did not wait for a bytecode or a MSIL to appear!

3. Toward a democratization of programming

We encounter a situation which is quite similar to the one born from the development of BASIC in the late 60's, which somehow led to the "personal computer revolution", the historic shift from corporate big computers to personal small ones with the ability given to programmers to interact with hardware using some general purpose operating system (DOS). BASIC language and DOS have appeared to

be insufficiently efficients and general to allow the development of "true" software applications on a window system. The same way the lack of unified approaches and of portability between different computers led to the development of several high-level languages, the lack of interoperability of software applications and the lack of coherence of software developments have led to a multitude of peripheral languages, mainly scripting and macro ones.

From this point of view, the main challenge of software developments stand in the simplification and standardization of environment and language interactions (it means user, system, network, application and object interactions). The standard interface used to be based on console, file sharing and API. But, nowadays, WEB/graphic interfaces, server/client, shared memory and messages seem to be the main interaction frameworks.

Thus, integration has more or less be solved by introducing standard libraries and intermediate machine code to perform the interface with the window manager and the operating system, while client/server protocols and messages are assumed to be a panacea for software cooperation. No need to say that each of these approaches assume the introduction of a specific language (bytecode, interface description language, server/client protocol, or whatever...) restricting *de facto* the depth of the interaction. Moreover, such techniques are only concerned with the integration with target language environments, and the so-built applications share all the characteristics of language-based applications: no coherence may be found in their internal representations.

Last, the integration of several modules within a single language framework goes through the encapsulation (object oriented programming, name scoping) to avoid implicit unwanted interactions, mainly: name clashes. The intrinsically flat model of resource naming of most of the compilers lead to incredibly complex construction to partially achieve this goal.

All of this is the mere consequence of an old vision of the compilation process, which tend to disappear with the aim of three main concepts: reflexivity, dynamic compilation and modularity, which give the possibility for the code to look at the internals of the compiler, to modify the parsing and/or compilation rules and to ask for the compilation of some new (generated) code, each of these being performed in some private local module. Interaction between modules is then solved by the description of the visibility of names from one module to another.

4. Toward a programming operating system?

A quick overview of existing languages displays a large common core of services, which might be enlarged in an object oriented view. Several attempts to build some "object operating system" failed, mainly because of the restriction of the prerogatives of such an operating system to a mere data management. According to a view of a "programming language" as a "way to say", a programming operating system should have very large attributions:

- **Data management.** It means memory management (cache, garbage collection, etc.) as well as data access resolution and control.
- **Parallelism.** Here, we mean the management of parallelism, multi-threading, message triggering and synchronization.
- **Algorithmic management.** This concerns the management of all the services more or less dependent of a particular data structure.
- **Name and modularity management.** It concerns resources naming, module encapsulation and visibility handling.

Altogether, if we want an operating system to fulfill all these requirements, it should also responsible for a part of the parsing and compilation. Therefore, it appears as a kind of "universal dynamic modular compiler".

Recall most of the compilers are concerned with two stages: the parsing (syntax and grammar recognition), which leads to the construction of an expression tree, and the compilation of the expression tree itself into an instruction list, which might be further optimized. A programming operating system should be responsible for both parts, according to local software-defined rules. A language is then defined as a set of parsing and compilation procedures.

Although basic rules might be defined using a core minimal language, user defined languages should also be able to express new rules (meta-programming). Last, imperative languages (like C), declarative languages (like PROLOG), functional ones (like CAML), languages without reserved keywords (like PL/I or REXX), as well as relational languages (like SQL) should have a possible definition. This way, languages are reduced to the organization of some conceptual framework into a coherent "way to say".

Frameworks may easily interact with each other, as data, functions and even abstractions like types are shared at programming operating system level. In order to avoid name clashes, compilation should use a locality principle: each "module" has some view (or map) of its programming environment, while the operating has (directly or indirectly) access to the whole atlas.

The multiplicity of frameworks allows to meet to paradoxical requirements of the programmer: ease to write and efficiency. A "catch-all" language seems an utopia to me and it is of a main importance to keep the ability for the programmer to solve such or such problem at the level of abstraction to which it corresponds (who would write a driver in ADA or a spreadsheet in assembly?). Contrary to the common opinion, global coherence is not incompatible with a strong adaptation to a specific field, but this coherence will need developers to be more mature and less centered on their own problematic.

This approach allows integration at language level. For instance, integration of a database manager and of an HTTP server allows the introduction in dynamic pages' code of operators involving heavy cooperation of these, in order to achieve, for instance, automatic database update with access rights checked through digital signatures. This approach is the one of Pliant project (http://pliant.cx).

(P. Ossona de Mendez) Centre d'Analyse et de Mathématiques Sociales, CNRS, UMR 8557, 54 Bd Raspail, 75006 Paris, France

E-mail address: pom@ehess.fr

URL: http://www.ehess.fr/centres/cams/person/pom/index.html

(H. Tonneau) Heliogroup company, 45 bis rue du sergent Bobillot, 93100 Montreuil, France

E-mail address: hubert.tonneau@heliosam.fr