Science of Computer Programming 97 (2015) 143–149

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Towards multilingual programming environments

Tijs van der Storm, Jurgen J. Vinju *

CWI, Amsterdam, Netherlands

ARTICLE INFO

Article history:
Received 8 November 2013
Accepted 28 November 2013
Available online 10 December 2013

Keywords:
Programming environments
Language interoperability
Metaprogramming

ABSTRACT

Software projects consist of different kinds of artifacts: build files, configuration files, markup files, source code in different software languages, and so on. At the same time, however, most integrated development environments (IDEs) are focused on a single (programming) language. Even if a programming environment supports multiple languages (e.g., Eclipse), IDE features such as cross-referencing, refactoring, or debugging, do not often cross language boundaries. What would it mean for programming environment to be truly multilingual? In this short paper we sketch a vision of a system that integrates IDE support across language boundaries. We propose to build this system on a foundation of unified source code models and metaprogramming. Nevertheless, a number of important and hard research questions still need to be addressed.

Programming environments are an important thread through Paul's research career. From the early papers on the ASF+SDF Meta-Environment to the Rascal programming language of today, a constant focus of Paul's work has been to improve the life of the programmer with better tools, better designs, and better languages. In the meantime the software landscape has only become more complex, more heterogeneous and more multi-faceted. This short paper envisions a programming environment that both embraces and unifies this multiplicity using one of Paul's favorite topics: metaprogramming. Thanks Paul, for our professional careers and for your friendship. We hope you enjoy reading this paper.

1. Introduction

Most software projects consist of many kinds of different artifacts, in different languages. For instance, a typical Java Web application project might contain Java source files, templates (e.g., JSP), Javascript source files, SQL schema definitions, ORM mapping files, and HTML templates. All these artifacts are related to each other. They may refer to each other through special names. For instance, class names coincide with database tables and HTML templates refer to tag libraries. Generally IDEs do not take the inter-operation of these languages into account; the reference links are not explicitly modeled, and thus not actionable, but they do exist in the code base. As a result, supporting features such as cross-referencing, refactoring, and debugging do not cross linguistic boundaries. This leads to inaccurate support which eventually leads to bugs that must be resolved later.

Another source of language multiplicity is the meta information that is necessary to build, configure and deploy projects: build files (e.g., Ant, Maven), configuration files (e.g., Spring XML files). These languages seem secondary, but they have big impact on the semantics of the eventually running program, and as such may contain bugs. More problematically, language support for the other languages (such as Java) does not model most of the information in this meta data, introducing inaccuracy in IDE features such as quick lookup, flow graphs and autocomplete. The more frameworks that offer reuse and high levels of abstraction are introduced, the harder it becomes to provide meaningful IDE features.

* Corresponding author.
E-mail addresses: storm@cwi.nl (T. van der Storm), jurgenv@cwi.nl (J.J. Vinju).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.041

Language referencing and meta information are just two examples of relations between languages. Other relations between languages include containment, where one language is embedded in another (e.g., SQL in COBOL code), – or derivation where one language is compiled into another language (e.g., code generation of a DSL). We expect truly multilingual IDEs to take all of these relations into account.

Common IDEs like Eclipse provide IDE support for some of these languages and some of their combinations. They are, however, primarily targeted at a single programming language and there exists ample opportunity for further integration. For instance, Eclipse is mainly an IDE for Java. The plugin system of Eclipse allows users to get IDE support for other languages, such as JavaScript, XML, SQL, etc. However, such plugins are mostly isolated from each other: integration across language boundaries is limited. Often, each language lives in its own silo. As a result, the programmer constantly has to switch perspectives and mentally keep the different artifacts in sync.

In this paper we analyze this problem in some depth and propose an approach to overcome these limitations. The key aspect of our design is to consider the multiplicity of languages as a federation of languages. Only the combination of all artifacts leads to the software system captured by the project, – in a sense there is only one big, composite language. Understanding a software project thus means understanding how these languages work together. Consequently, we hope, truly multilingual IDE support will be instrumental in improving understanding and thus in helping to construct and maintain high quality software.

2. Towards monolingual programming environments, redux

Heering and Klint wrote “Towards monolingual programming environments” in 1985 [8], warning us for the complexity of the exploding number of languages in programming environments and proposing to fully reverse this development into a single language with a single and consistent programming and debugging environment. Today we are faced with what we were warned for: hundreds of independent languages for programming, scripting, configuring, defining, and debugging software. Moreover, due to the availability of memory and disk space, we now have all these languages installed and active within the same computer system. For the sake of argument, let us assume this complex reality was introduced for all good reasons.

In this paper we propose to view the de facto multiplicity of languages that a programmer is subjected to as a single, federated language. This federation of languages encompasses all kinds of “source code in the broad sense” [18]. What is the syntax of this language? What is its semantics? How do we model name resolution, declarations, uses, control flow, data flow, and types for this language? Given answers to these questions, we will have a principled method of modeling cross language semantic dependency. On top of such generic models, advanced IDE features such as refactoring tools, debugging, hover help, reference hyperlinks and auto-complete may be constructed. We propose a high level design and a research agenda towards integrated, multi-lingual development environments.

2.1. One IDE to rule them all

Here we describe the high-level requirements for enabling the syntactic and semantic integration of IDE support for multiple software languages. This design acts as a frame of reference for identifying the open problems that we discuss in Section 3.

Uniform representation. At a very basic level language artifacts are a form of structured data. To allow this data to be processed in a typed and uniform way, requires three meta-level services. First, the structure of the data should be modeled in a uniform way, for instance using meta modeling, data description, or schema language. Second, the data itself should have a uniform, typed in-memory representation. Finally, a typed serialization/deserialization service is needed to load the language artifacts to/from disk.

Uniform identification. Artifacts and sub-entities of those artifacts often have an identity to be able to refer to them. When a project consists of multiple languages, there are multiple meta models at play. Through the uniform representation the artifacts can be processed in a uniform way. However, each language might provide specific ways of identifying entities. To refer to elements from different languages in a uniform way we need a generic identification mechanism that works for all of them.

Modeling relations. There are many examples of possible relations between artifacts and relations between sub-artifact elements: containment, call graphs, use-define relations, import relations, control-flow and data flow graphs, inheritance relations etc. It should be possible to super-impose such relations on top of the uniform representation of the artifacts. The generic identification mechanism plays a key role here.

Hooks into the user interface. The analyses and transformations realized on top of the uniform representations of artifacts and the relations between them should be made available to the programmer through user interface affordances. The key requirement, however, is that the interface is language parametric. It should be possible to, for instance, provide syntax coloring for a language without having to customize the IDE per language. The same holds for invoking refactorings, hyperlinking artifacts, creating outlines etc.