Science of Computer Programming 95 (2014) 219–253

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A meta-circular language for active libraries

Marco Servetto [a], Elena Zucca [b,*]

[a] Victoria University of Wellington, New Zealand
[b] DIBRIS, Università di Genova, Italy

HIGHLIGHTS

- We present a new Java-like language design coupling disciplined meta-programming features with a composition language.
- We provide an effective language support for active libraries.
- An incremental approach prevents type errors, while keeping expressiveness.
- Meta-level soundness is ensured, that is, programmers can safely use active libraries.

ARTICLE INFO

Article history:
Received 22 May 2013
Received in revised form 12 May 2014
Accepted 14 May 2014
Available online 27 May 2014

Keywords:
Java
Meta-programming
Nested classes
Active libraries

ABSTRACT

We present a new Java-like language design coupling disciplined meta-programming features with a composition language. That is, programmers can write meta-expressions that combine class definitions, on top of a small set of composition operators, inspired by the seminal Bracha's Jigsaw framework. Moreover, such operators are *deep*, that is, they allow manipulation (e.g., renaming or duplication) of a nested class at any level of depth. This provides an effective language support for *active libraries*: namely, a (library) class can provide a method returning a customized version of a class, depending, e.g., on the execution platform. Since a class can contain nested classes, a whole library can be generated in this way. That is, deep operators allow the programmer to better exploit meta-programming capabilities, leading to a "meta-programming in the large" style. We adopt a mixed typechecking technique, which provides a good compromise between meta-programming systems with extreme expressiveness and no static type checking, and those with strong type system and only limited meta-programming capability. In particular, our technique ensures an important property, called *meta-level soundness*, stating that typing errors never originate from already compiled (meta-)code, that is, programmers can safely use (active) libraries.

1. Introduction

Differently from conventional libraries, which offer a fixed set of data types and operations, *active libraries* [25] are parametric, in the sense that the user can obtain a customized version of the library code by providing values for some parameters.

The process of producing the concrete version of the library starting from the parametrized one is often called *instantiation* of the library. During such a process, the library can interact dynamically with the compiler, providing meaningful error messages or platform-specific optimizations.

* Corresponding author.

http://dx.doi.org/10.1016/j.scico.2014.05.003
0167-6423/

Code reuse through inheritance is, in a sense, the ancestor of active libraries: indeed, a (library) class where well-documented virtual methods can be overridden, in order to provide a specialized version of the class itself, serves the role of an active library, even though with great expressiveness limitations. From this point of view, all the subsequent research on going beyond inheritance by more flexible composition mechanisms, e.g., [9,27,31,40,51], has aimed at reducing such limitations, by balancing expressivity, (type-)safety, maintenability and a straightforward semantics.

A more powerful approach is offered by *meta-programming*, which allows programmers to write (meta-)code that can be used to directly generate customized code, rather than incrementally extending existing code. While meta-programming maximises expressivity, usually it offers no guarantee of (type-)safety, maintainability and simple semantics. Only some approaches, notably MetaML [74], Java Mint [77] and MorphJ [37], offer type safety at the price of statically fixing the structural shape of the resulting class.

In the context of Java-like languages, the most widely used technique to support active libraries is C++ *template meta-programming* [72]. In C++, templates are parametric functions or classes that can be defined and later instantiated to obtain highly optimized specialized versions. This technique is very powerful, yet can be difficult to understand, since its syntax and idioms are esoteric compared to conventional programming. More recently, Template Haskell [69] and F# type providers [73] have offered a similar expressive power, but using a meta-circular approach: that is, syntax and idioms for instantiating the active library are the same of the conventional language.

However, maintaining and evolving code which exploits this meta-programming technique is rather complex, and hard to debug, since well-formedness of generated source code can be checked only "a posteriori". That is, every time we instantiate a template, client code and template instantiation must be typechecked together. Hence, the template code needs to be re-analyzed many times, and the client can get error messages which do not depend on the code she has written.

Despite all these limitations, template meta-programming is widely used, proving that there is a strong need for its expressive power.

In [67] we have proposed a new language design, called MetaFJig, for Meta Featherweight Jigsaw, which provides a good compromise between systems with extreme expressiveness and no static type checking, and those with strong type system and only limited meta-programming capability. This compromise is achieved in two ways:

- The programmer can write meta-code which generates new classes, but only by combining handwritten classes by a given set of composition operators.
- Type safety is ensured by a mixed technique, called *checked compile-time execution*, where Java errors are detected by the standard compiler, whereas composition errors, e.g., summing two classes with conflicting members, are detected dynamically, that is, are exceptions, which can be handled by the programmer.

In particular, our approach ensures the important property, which we call *meta-level soundness*, that, contrarily to what happens in template meta-programming, typing errors never originate from already compiled (meta-)code, hence programmers can safely use (active) libraries. In other words, meta-level soundness is just the standard requirement of *compositionality* (in order to safely compose library and client code it is enough to typecheck client code against the library), which, however, in the meta-programming case is not taken as granted.

In this paper we present MetaFJig*, an extended version of MetaFJig [67] where classes can be nested and composition operators are *deep*, as we have introduced in some previous work not related to meta-programming [22–24].

The combination of these two orthogonal features (meta-programming and deep operators) turns out to be very useful. Indeed, the expressive power of the meta-level, together with the capability of representing a whole component as a single class, allows one to encapsulate a library within a single meta-expression. Moreover, the possibility offered by the meta-level to write classes whose structure depends on an external source, like a database table, can be generalized to a whole hierarchy, as one can extract from a whole database or XML schema. Significant examples which can be expressed in MetaFJig* and were not expressible in our previous work are shown in Section 3.

From the technical point of view, combining the two features provides, in a sense, a test on the modularity of our approach. Despite the much greater complexity of the language with nesting, we did not discover any conceptual inadequacy of our meta-level construction, thus supporting our belief that its principles could be successfully applied to different real languages. However, the combination posed some interesting design issues, notably concerning the dependencies among classes during checked compile-time execution, as we will better explain in Section 4.

This paper is a revised and largely extended version of [68], where the new contributions are the formal definition of checked compile-time execution for MetaFJig* and the related results, more examples and a detailed discussion of related work.

To make the presentation lighter, we only formalize a minimal subset of the language, which has been chosen to illustrate all the essential features of the model. The formal definition for the whole language can be found in first author's PhD thesis [66].

A prototype compiler, supporting a superset of MetaFJig*, can be downloaded (along with its sources and some examples) at: `homepages.ecs.vuw.ac.nz/~servetto/MetaFJig/`. All the examples shown in this paper are supported by the prototype.

The rest of the paper is organized as follows: in Section 2 we informally present MetaFJig* by examples, and in Section 3 we provide more significant examples showing the expressive power. In Section 4 we illustrate by some examples how