Customizable composition operators for Java-like classes (extended abstract)

Giovanni Lagorio, Marco Servetto, and Elena Zucca

DISI, University of Genova, Italy

Abstract. We propose a formal framework for extending a class-based language, equipped with a given class composition mechanism, to allow programmers to define their own derived composition operators. These definitions can exploit the full expressive power of the underlying computational language. The extension is obtained by adding *meta-expressions*, that is, (expressions denoting) class expressions, to conventional expressions. Such meta-expressions can appear as class definitions in the class table.
Extended class tables are reduced to conventional ones by a process that we call *compile-time execution*, which evaluates these meta-expressions. This mechanism poses the non-trivial problem of guaranteeing soundness, that is, ensuring that the conventional class table, obtained by compile-time execution, is well-typed in the conventional sense. This problem can be tackled in many ways. In this paper, we illustrate a lightweight solution which enriches compile-time execution by partial typechecking steps.

Introduction

Support for code reuse is a key feature which should be offered by programming languages, in order to automate and standardize a process that programmers should, otherwise, do by hand: duplicating code for adapting it to best solve a particular instance of some generic problem. Two different strategies which can be adopted to achieve code reuse are *composition languages* and *meta-programming*.
In the former approach programmers can write fragments of code (classes in the case of Java-like languages) which are not self-contained, but depend on other fragments. Such dependencies can be later resolved by combining fragments via composition operators, to obtain different behaviours. These operators form a *composition language*. Inheritance (single and multiple), mixins and traits are all approaches allowing one to combine classes, hence they define a composition language in the sense above.
The limitation of this approach is that the users, provided with a fixed set of composition mechanisms, cannot define their own operators, as it happens, e.g., with function/method definitions.
In meta-programming, programmers write (meta-)code that can be used to generate code for solving particular instances of a generic problem. In the context of Java-like languages, *template meta-programming* is the most widely used meta-programming facility, as, e.g., in C++, where templates, which are parametric functions or classes, can

be defined and later instantiated to obtain highly-optimized specialized versions. The instantiation mechanism requires the compiler to generate a temporary (specialized) source code, which is compiled along with the rest of the program. Moreover, template specialization allows to encode recursive computations, that can be thought of as compile-time executions. This technique is very powerful, yet can be very difficult to understand, since its syntax and idioms are esoteric compared to conventional programming. For the same reasons, maintaining and evolving code which exploits template meta-programming is rather complex, see, e.g., [1]. Moreover, well-formedness of generated source code can only be checked "a posteriori", making the whole process hard to debug.

Here, our aim is to distill the best of these two approaches, that is, to couple disciplined meta-programming features with a composition language, in the context of Java-like classes. More precisely, we propose a formal framework for extending a class-based language, equipped with a given class composition mechanism, to allow programmers to define their own derived composition operators. These definitions can exploit the full expressive power of the underlying computational language.

The extension is obtained as follows: *meta-expressions*, that is, (expressions denoting) class expressions, are added to conventional expressions. Then, such meta-expressions can appear as class definitions in the class table. Extended class tables are reduced to conventional ones by evaluating these meta-expressions. This meta-circular approach implies compile-time execution as in template meta-programming, with the advantage of a familiar meta-language, since it just coincides with the conventional language the programmers are used to.

This mechanism, which is trivial in itself, poses the non-trivial problem of guaranteeing soundness, that is, ensuring that the conventional class tables, obtained by compile-time execution, are well-typed (in the conventional sense).

Ideally, typing errors in generated source code should be detected statically, that is, without requiring reduction at the meta-level at all, as it happens, e.g., in MetaML [10]. However, this would require to introduce sophisticated types for meta-expressions. In this paper, we illustrate instead a lightweight solution which enriches compile-time execution by typechecking steps. Conventional typechecking of class expressions only takes place when they appear as the right-hand side of class definitions in the class table. With this approach, it suffices to introduce a unique common type **`code`** for meta-expressions, at the price of a later error detection.

The paper is organized as follows: in Section 1 we informally introduce our approach by means of some examples, in Section 2 we provide a formalization on a very simple class composition language, and in Section 3 we summarize the contribution of the paper and draw related and further work. A full version of this paper is [5].

1 Examples

To add a meta-programming facility to a class-based language, we allow class (composition) expressions to be used as expressions of a newly introduced type: **`code`**.

For instance, the following program

```
class C = {
```

```
  code m() {
    return { int one() { return 1; }  };
  }
}
class D = new C().m()
```

declares two classes, `C` and `D`. The former, `C`, declares a single method named `m`, which returns a value of type **code**. This value, in turn, is a base class declaring the (non-meta) method[1] `one`. The latter class, `D`, is declared using an expression that has to be evaluated in order to obtain the corresponding class body.
One very basic use of this mechanism allows to obtain conditional compilation. For instance, in the previous example we could have written:

```
class C = {
  code m() {
      if (DEBUG) return /* ...debug version... */;
      return /* ...as before... */;
  }
}
```

The following (meta-)method:

```
code mixin(code parent) {
  return { /* ... */ } extends parent;
}
```

behaves like a mixin, extending in some way a parent class passed as argument.
Note that the code in the extension can select arbitrary fields or methods of the parent class. This is allowed because we do not typecheck a class expression until it is associated to a class name in the class table. This choice allows for an incredible leeway in writing reusable code, at the price of a late error detection. The situation is very similar to what happens with C++ templates [9,7].
The class to be used as parent could be constructed, having a generic list type, `List<>`, by chaining an arbitrary number of classes:

```
code chain(List<code> parents){
  if (parents.isEmpty()) return Object;
  return parents.head() extends this.chain(parents.tail());
}
```

This is similar to mixin composition, with the advantage that the operands of this arbitrarily long composition do not have to be statically known. Other examples can be found in the full version of this paper [5].

2 Formalization

We assume a conventional language in the style of Featherweight Java, as outlined in in Figure 1 .

[1] We call *meta-methods* the methods involving code manipulation.

$$
\begin{array}{lll}
cp & ::= \overline{\texttt{class}\ C\ = ce} & \text{(conventional) program} \\
ce & ::= C \mid B \mid ce\ \texttt{extends}\ ce' \mid \ldots & \text{class expression} \\
\ldots & & \\
e & ::= x \mid \texttt{new}\ C\ (\overline{e})\ \mid \ldots & \text{(runtime) expression} \\
v & ::= \texttt{new}\ C\ (\overline{v}) & \text{value} \\
T & ::= C & \text{type} \\
CT & ::= \ldots & \text{class type} \\
\Delta & ::= \overline{C{:}CT} & \text{class type environment} \\
\Gamma & ::= \overline{x{:}T} & \text{parameter type environment}
\end{array}
$$

Fig. 1. Syntax and types of the conventional language

Figure 2 shows how the conventional language is extended to allow customizable composition operators. As already mentioned, this is achieved by two steps: first, *meta-expressions*, that is, (expressions denoting) class expressions, are added to conventional expressions, as shown in the second production. These meta-expressions have a special primitive type **code** which is added to types (fourth production, and typing rules in the last section of the figure). In particular, a class expression is seen as a value of type **code** (third production). Moreover, such meta-expressions can appear as class definitions in the program (first production).

$$
\begin{array}{lll}
p & ::= \overline{\texttt{class}\ C\ = e} & \text{(generalized) program} \\
e & ::= \ldots \mid C \mid B \mid e\ \texttt{extends}\ e' & \\
v & ::= \texttt{new}\ C\ (\overline{v})\ \mid ce & \\
T & ::= C \mid \texttt{code} &
\end{array}
$$

$$
\text{(META-RED)}\ \frac{e \xrightarrow{cp} e'}{cp\ (C = e)\ p \longrightarrow cp\ (C = e')\ p}
$$

$$
\text{(EXTENDS-1)}\ \frac{e_1 \xrightarrow{cp} e_1'}{e_1\ \texttt{extends}\ e_2 \xrightarrow{cp} e_1'\ \texttt{extends}\ e_2}
\qquad
\text{(EXTENDS-2)}\ \frac{e \xrightarrow{cp} e'}{v\ \texttt{extends}\ e \xrightarrow{cp} v\ \texttt{extends}\ e'}
$$

$$
(\leq\text{-REFL-CODE})\ \frac{}{\Delta \vdash \texttt{code} \leq \texttt{code}}
\qquad
\text{(T-NAME)}\ \frac{}{\Delta; \Gamma \vdash C{:}\texttt{code}}
\qquad
\text{(T-BASIC)}\ \frac{}{\Delta; \Gamma \vdash B{:}\texttt{code}}
$$

$$
\text{(T-EXTENDS)}\ \frac{\Delta; \Gamma \vdash e_1{:}\texttt{code} \quad \Delta; \Gamma \vdash e_2{:}\texttt{code}}{\Delta; \Gamma \vdash e_1\ \texttt{extends}\ e_2{:}\texttt{code}}
$$

Fig. 2. Meta-expressions and compile-time execution

Then, *compile-time execution* consists in reducing this (generalized) program to a conventional program, where all right-hand sides of class declarations are values, that is, class expressions. This is modeled by the relation $p \longrightarrow p'$, whose steps are *meta-reduction* steps, that is, steps of reduction of a meta-expression. More precisely, as formalized by rule (META-RED), in a (generalized) program it is possible to reduce the right-hand-side of a class declaration in the context of a conventional fragment cp of the program. We have assumed (without any loss of generality) that in a generalized program the conventional part comes first. The relation $e \xrightarrow{cp} e$ is the standard FJ reduction of an expression in the context of a (conventional) program, enriched by the rules (**EXTENDS**-1) and (**EXTENDS**-2).
To guarantee that compile-time execution always produces a well-typed program when terminates, we can take different approaches. In this paper, we propose a simple technique which integrates meta-reduction with typechecking, as shown in Figure 3. In this approach, reduction of a program involves some typechecking steps, which can either succeed or fail. In the latter case the program reduces to `error`.

$$\text{(META-RED)}\ \frac{e \xrightarrow{cp} e'}{cp{:}\Delta\ cp'{:}\mathbf{code}\ (\mathbf{class}\ C = e{:}\mathbf{code})\ p \longrightarrow cp{:}\Delta\ cp'{:}\mathbf{code}\ (\mathbf{class}\ C = e'{:}\mathbf{code})\ p}$$

$$\text{(META-CHECK)}\ \frac{}{cp{:}\Delta\ cp'{:}\mathbf{code}\ (\mathbf{class}\ C = e)\ p \longrightarrow cp{:}\Delta\ cp'{:}\mathbf{code}\ (\mathbf{class}\ C = e{:}\mathbf{code})\ p}\ \Delta; \emptyset \vdash e{:}\mathbf{code}$$

$$\text{(META-CHECK-ERROR)}\ \frac{}{cp{:}\Delta\ cp'{:}\mathbf{code}\ (\mathbf{class}\ C = e)\ p \longrightarrow \texttt{error}}\ \begin{array}{l}\nexists cp'' \subseteq cp',\ cp'' \neq \emptyset \text{ s.t. } closed(\Delta, cp'') \\ \Delta, \emptyset \nvdash e{:}\mathbf{code}\end{array}$$

$$\text{(CHECK)}\ \frac{}{cp{:}\Delta\ cp'{:}\mathbf{code}\ \tilde{p} \longrightarrow cp{:}\Delta\ cp'{:}\Delta'\ \tilde{p}}\ \begin{array}{l}cp' \neq \emptyset \\ \Delta \vdash cp'{:}\Delta'\end{array}$$

$$\text{(CHECK-ERROR)}\ \frac{}{cp{:}\Delta\ cp'{:}\mathbf{code}\ \tilde{p} \longrightarrow \texttt{error}}\ \begin{array}{l}cp' \neq \emptyset \\ closed(\Delta, cp') \text{ or } \tilde{p} = \emptyset \\ \nexists \Delta'.\Delta \vdash cp'{:}\Delta'\end{array}$$

Fig. 3. Checked compile-time execution

More in detail, during compile-time execution each class declaration **class** $C = e$ in the program can be annotated with the following meaning:

- empty annotation: initial state, no check has been performed yet;
- annotation **code**: e is a well-typed meta-expression;
- annotation CT, for some class type CT: e is (a well-typed meta-expression which denotes) a well-typed class expression of type CT.

We will use $\tilde{p}$ as metavariable for annotated programs. More precisely, checked compile-time execution is defined on annotated programs of the following form:
$\tilde{p} ::= cp{:}\Delta\ cp'{:}\mathbf{code}\ [\mathbf{class}\ C = e{:}\mathbf{code}]\ p \mid \texttt{error}$

where square brackets denote optionality, and e is not of the form ce. Moreover, for any cp conventional program, cp:**code** is the program obtained by annotating each class declaration by **code**, and, for any Δ s.t. $dom(cp) = dom(\Delta)$, cp:Δ is the program obtained by annotating each class declaration with the type associated in Δ to the corresponding class name.
We have assumed (without any loss of generality) that in an annotated program the cp:Δ part comes first, then the cp:**code** part, then the others. In particular, in the initial program conventional class declarations appear first and are annotated **code**. Moreover, reduction rules ensure that at each intermediate step there is at most one class declaration which has been annotated **code** but is not reduced yet (this is formalized by the subject reduction property).
Rule (META-RED) models a (safe) meta-reduction step. Indeed, meta-reduction is only performed w.r.t. a conventional program cp which has been previously succesfully typechecked. Note that, here as in the following two rules, there can be another portion of the program cp' which has already been reduced, but for which it is still impossible to perform a conventional typechecking step. This happens when cp' refers to some class names whose definition is still unavailable.
Rule (META-CHECK) and (META-CHECK-ERROR) model a typechecking step at the meta-level. That is, the first class declaration in the program which is not annotated yet is examined, to check that its right-hand side e is a well-typed meta-expression. The expression is typechecked w.r.t. to the portion of the conventional program cp which has been already successfully typechecked. If the typechecking step succeeds, then the class declaration is annotated **code**. Otherwise, an error is raised only if it is not possible to perform a further conventional typechecking step on cp', since any non-empty subset of cp' refers to some class names whose definition is still unavailable. This is expressed by the side-condition: $closed(\Delta, p)$ holds when p only refers to class names that are either in $dom(\Delta)$ or in $dom(p)$ itself (the trivial formal definition is omitted).
Rule (CHECK) and (CHECK-ERROR) model a conventional typechecking step. A successful typechecking step takes place if there is a portion of the conventional program cp' which can be typechecked w.r.t. the current class type environment Δ. An error is raised, instead, if no successful typechecking step is possible and, moreover, there is no hope it will be possible in the future, since either cp' only refers to class names which are already available, or there are no other class definitions to reduce.
Examples of reductions can be found in [5]. Soundness is formally expressed by the usual progress and subject reduction properties, which are also proved in [5].

3 Related work

Metaprogramming approaches can be classified by two properties: whether the meta-language coincides with the conventional language (the so-called *meta-circular* approach), and whether the code generation happens during compilation. MetaML [10], Prolog [8] and OpenJava [11] are meta-circular languages, while C++ [4], D [2], Metatrait-Java [6] and MorphJ [3] use a specialized meta-language.[2] Almost any dynamically typed language allows some sort of meta-circular facility, typically by offering

[2] The latest version of D seems to include a limited form of metacircular compilation.

an *eval* function. Such a function allows to run arbitrary code, represented by an input string. Regarding code generation, MetaML and Prolog performs the computation at run time, while C++, D, Meta-trait-Java, MorphJ and OpenJava use compile-time execution. Again, dynamically typed languages providing an *eval* function allow runtime meta-programming. The work presented in this paper lies in the area of meta-circular compile-time execution. Among the above mentioned approaches, [11] is the one showing more similarities with ours. OpenJava offers the ability to define new language constructs, on top of Java, using meta-circular compile-time execution. Programmers can define new constructs by writing *meta-classes*, that is, particular Java classes which instruct the OpenJava compiler on how to perform the type-driven translation. These meta-classes use the reflection-based *Meta Object Protocol (MOP)* to manipulate the source code and provide its translation. However, their approach is definitely lower level than ours and we have a very different long-term goal: that is, to bring compile-time execution in the realm of an already familiar programming language, rather than to allow programmers to define their own extensions of an existing language.

References

1. Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*. Springer, 2004.
2. Digital Mars. D programming language, 2007. http://www.digitalmars.com/.
3. Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *ECOOP'07 - Object-Oriented Programming*, pages 399–424. Springer, August 2007.
4. International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, 2003.
5. Giovanni Lagorio, Marco Servetto, and Elena Zucca. A lightweight approach to customizable composition operators for Java-like classes. In *FACS'09 - International Workshop on Formal Aspects of Component Software*, 2009.
6. John Reppy and Aaron Turon. Metaprogramming with traits. In Erik Ernst, editor, *ECOOP'07 - Object-Oriented Programming*, number 4609 in Lecture Notes in Computer Science, pages 373–398. Springer, 2007.
7. Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Bern, February 2005.
8. Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, April 1994.
9. Bjarne Stroustrup. *The C++ Programming Language*. Reading. Addison-Wesley, special edition, 2000.
10. Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.
11. Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Kilijian, and Kozo Itano. OpenJava: A class-based macro system for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science, pages 117–133. Springer, 2000.