Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 263 (2010) 161–177
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Lightweight Approach to Customizable Composition Operators for Java-like Classes[1]

Giovanni Lagorio[2], Marco Servetto[3] and Elena Zucca[4]

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
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. Conventional typechecking of class expressions only takes place when they appear as 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.

Keywords: Modular composition, Java-like languages, Meta-programming, Type systems

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*.

[1] This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

[2] Email: lagorio@disi.unige.it

[3] Email: servetto@disi.unige.it

[4] Email: zucca@disi.unige.it

doi:10.1016/j.entcs.2010.05.010

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 [5], as, e.g., in C++, where templates, which are parametric functions or classes [6], 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., [3]. 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 [7] 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

[5] A very limited form of meta-programming is offered by Java/C# generics, which are classes parametric in some type parameters, that can be compiled once. Java generics are a source level feature, which is compiled away, while generics are fully supported by C#.

[6] Note that C++ supports, along type-parameters, other kinds of template parameters.

[7] In this paper for sake of simplicity we illustrate the approach on only one composition operator, that is, `extends`.

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 [17]. 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. The Appendix contains the proof of soundness. This paper is the full version of [8].

1 Examples

In order to show how to use meta-programming as a tool for better composing software, we introduce a language allowing one to compose classes by means of some operators. In such a language, a class declaration associates a *class expression* with the name of the declared class. The simplest form of class expression is the *base class*, that is, a set of field and method declarations. For instance, the literal `{ int answer() { return 42; } }` denotes a base class declaring a single method named `answer`. In our example language, we can give the name `C` to that class body by writing:

```
class C = {  int answer() { return 42; }  }
```

This is the exact Java syntax, with the exception of the extraneous symbol `=`. Of course, here the symbol `=` can be followed by any arbitrarily involved class composition expression, in place of a simple base class literal.

Since our aim here is to explain how our approach works, rather than proposing a specific composition language, for simplicity we consider a very simple language offering just a single binary operator, `extends`, allowing one to combine two classes in a way that should feel natural to Java programmers: the left operand *extends*, that is, overrides, the right operand.

For instance, writing

```
{ int a() { return 1; } } extends
{ int a() { return 0; } int b() { return 0; } }
```

is equivalent to write:

```
{ int a() { return 1; } int b() { return 0; } }
```

To add a meta-programming facility to this simple 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[8] `one`. The latter class, `D`, is declared using an expression that has to be evaluated in order to obtain the corresponding class body. In this example, the body of `D` is the value returned by the method `m` of `C`, so this program could be equivalently written as:

```
class C = /* ...as before... */
class D = {  int one() { return 1; }  }
```

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 [16,14].

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 indeed similar to mixin composition, with the advantage that the operands of this arbitrarily long composition do not have to be statically known.

Finally, the following example is a graphic library that adapts itself with respect to its execution environment, without requiring any extra-linguistic mechanisms:[9]

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

[9] To keep the example compact, we do not detail all the classes named in the example, and we simply assume that they are declared elsewhere.

```
class GraphicalLibrary {
  code produceLibrary() {
    code result = BaseGraphicalLibrary;
    String producer = System.getProperty("sys.vcard.brand");
    if (producer.equals("NVIDIA"))
        result = NVIDIASupport extends result;
    else if (producer.equals("ATI"))
        result = ATISupport    extends result;
    else
        throws new CompilationError(
                         "No compatible hardware found");
    if (System.getProperty("os.name").contains("Windows"))
      result = CygwinAdapter extends result;
    return result;
  }
}
```

The method `produceLibrary` builds a platform-specific library by combining the generic library `BaseGraphicalLibrary` with the brand-specific drivers (represented by the two classes `NVIDIASupport` and `ATISupport`) and wrapping the result, if required on the specific platform, with the class `CygwinAdapter`, which emulates a Linux-like environment on Windows operating systems.

In this way the compilation of the same source produces customized versions of the library depending on the execution platform. In other words, this approach can be used to write *active libraries* [2], that is, libraries that interact dynamically with the compiler, providing better services, as meaningful error messages, library-specific optimizations and so on.

2 Formalization

Figure 1 shows syntax, values and types of our conventional language, using the overbar notation to denote a (possibly empty) sequence. [10]

The top section of the figure defines the syntax, where we assume infinite sets of class names C, field names f and method names m. As already mentioned, to keep the presentation minimal we consider a class composition language with only one operator, `extends`. This conventional language is very similar to Featherweight Java [6], FJ for short, but the operator `extends` composes two class expressions, rather than the name of an existing class with a class body (base class).

The semantics of the conventional language is given in the standard small-steps operational style [6,12], that is, by defining a reduction relation $e \xrightarrow{cp} e'$ on (run-time) expressions in the context of a program and a subset of expressions called *values* which correspond to final results of the evaluation and play a role in the evaluation strategy.

Reduction rules are as in FJ and are omitted. The only difference is that look-up, formally expressed by the function *mbody*, needs to be generalized, as shown in Figure 2, to take into account that `extends` composes two class expressions. We omit the analogous trivial generalization of the function *fields*. Values v of the conventional language are as in FJ. Note that this definition of values is recursive but well-founded, since the basis of the inductive definition are values of form `new` C`()`,

[10] This notation corresponds to the Kleene-star used in BNF-style.

cp	$::= \overline{\texttt{class}\ C\ = ce}$	(conventional) program
ce	$::= C \mid B \mid ce\ \texttt{extends}\ ce'$	class expression
B	$::= \{fds\ mds\}$	base class
fds	$::= \overline{fd}$	field declarations
fd	$::= T\ f\texttt{;}$	field declaration
mds	$::= \overline{md}$	method declarations
md	$::= T\ m(\overline{T\ x})\ \{\texttt{return}\ e;\}$	method declaration
e	$::= x \mid e.f \mid e.m(\overline{e}) \mid \texttt{new}\ C(\overline{e}) \mid (C)e$	(runtime) expression
v	$::= \texttt{new}\ C(\overline{v})$	value
T	$::= C$	type
CT	$::= \langle \overline{C}, fds, mhs \rangle$	class type
mhs	$::= \overline{mh}$	method headers
mh	$::= T\ m(\overline{T})$	method header
Δ	$::= \overline{C{:}CT}$	class type environment
Γ	$::= \overline{x{:}T}$	parameter type environment

Figure 1. Syntax and types of the conventional language

that is, objects with no fields.

Typing rules are shown in Figure 2.

The first four rules define the subtyping relation. Note that, since a class definition can contain many class names as subterms[11], in our generalization a class can be a direct subtype of many others. However, method look-up function *mbody* gives precedence to the left operand as in standard FJ.

Rule (METHOD-T) is as in FJ, typing rules for expressions are also as in FJ and are omitted.

The typing judgment $\Delta; C \vdash ce{:}\langle \overline{C}, fds, mhs \rangle$ assigns a class type to a class expression ce appearing as (subterm of) the definition of class C, needed to type method bodies in base classes in ce. This class type models the type information which can be extracted from ce, and consists of three components: a set $\overline{C}$ of class names (those appearing as subterms in ce, which are, hence, the direct supertypes of C), a set of field declarations and a set of method headers extracted from method declarations. As usual, we assume that these sets are well-formed only if a field (method) name appears only once, and write *dom* to denote the set of declared names. In rule (EXTENDS-T), this assumption implicitly ensures that a method can be overridden only with the same type, whereas the additional side condition prevents hiding of fields, and both are standard FJ requirements.

[11] For instance, `class C = D extends E`.

$$(\text{<-DIRECT})\ \frac{}{\Delta\vdash C{<}C_i}\ \begin{array}{l}\Delta(C)=\langle C_1\ldots C_n,\_,\_\rangle\\ i\in 1..n\end{array}\qquad (\text{<-TRANS})\ \frac{\begin{array}{c}\Delta\vdash C{<}C'\\ \Delta\vdash C'{<}C''\end{array}}{\Delta\vdash C{<}C''}$$

$$(\le\text{-REFL})\ \frac{}{\Delta\vdash C{\le}C}\qquad (\le\text{-STRICT})\ \frac{\Delta\vdash C{<}C'}{\Delta\vdash C{\le}C'}$$

$$(\text{METHOD-T})\ \frac{\begin{array}{l}\Delta;x_1{:}T_1,\ldots x_n{:}T_n,\texttt{this}{:}C\vdash e{:}T'\\ \Delta\vdash T'{\le}T\end{array}}{\Delta;C\vdash T\ m(T_1\ x_1\ldots T_n\ x_n)\{\texttt{return}\ e;\}{:}T\ m(T_1\ldots T_n)}$$

$$(\text{NAME-T})\ \frac{}{\Delta;C'\vdash C{:}\langle C,\mathit{fds},\mathit{mhs}\rangle}\ \Delta(C)=\langle\_,\mathit{fds},\mathit{mhs}\rangle$$

$$(\text{BASIC-T})\ \frac{\Delta;C\vdash md_1{:}mh_1\ldots\Delta;C\vdash md_n{:}mh_n}{\Delta;C\vdash\{\mathit{fds}\ md_1\ldots md_n\}{:}CT}\ CT=\langle\emptyset,\mathit{fds},mh_1\ldots mh_n\rangle$$

$$(\text{EXTENDS-T})\ \frac{\begin{array}{c}\Delta;C\vdash ce_1{:}\langle\overline{C}_1,\mathit{fds}_1,\mathit{mhs}_1\rangle\\ \Delta;C\vdash ce_2{:}\langle\overline{C}_2,\mathit{fds}_2,\mathit{mhs}_2\rangle\end{array}}{\Delta;C\vdash ce_1\ \texttt{extends}\ ce_2{:}\langle\overline{C}_1{\cup}\overline{C}_2,\mathit{fds}_1{\cup}\mathit{fds}_2,\mathit{mhs}_1{\cup}\mathit{mhs}_2\rangle}\ dom(\mathit{fds}_1){\cap}dom(\mathit{fds}_2)=\emptyset$$

$$(\text{PROGRAM-T})\ \frac{\begin{array}{l}\Delta,\Delta';C_1\vdash ce_1{:}CT_1\\ \ldots\\ \Delta,\Delta';C_n\vdash ce_n{:}CT_n\end{array}}{\Delta\vdash C_1=ce_1\ldots C_n=ce_n{:}\Delta'}\ \begin{array}{l}\Delta'=C_1{:}CT_1\ldots C_n{:}CT_n\\ \Delta,\Delta'\vdash\_{<}\_\ \text{acyclic}\end{array}$$

$\mathit{mbody}_{cp}(C,m)=\mathit{mbody}_{cp}(ce,m)$ if $cp(C)=ce$
$\mathit{mbody}_{cp}(\{\ldots C\ m(\ldots)\{\texttt{return}\ e;\}\ldots\},m)=e$
$\mathit{mbody}_{cp}(ce_1\ \texttt{extends}\ ce_2,m)=\mathit{mbody}_{cp}(ce_1,m)$ if $\mathit{mbody}_{cp}(ce_1,m)$ defined,
 $\mathit{mbody}_{cp}(ce_2,m)$ otherwise

Figure 2. Typing rules and look-up of the conventional language

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

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

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

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

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

Figure 3. Meta-expressions and compile-time execution

In rule (PROGRAM-T), standard FJ typing rule for programs is generalized to open programs, that is, programs which can refer to already compiled classes, modeled by the left-side class type environment Δ. We denote by Δ, Δ' concatenation of two class type environments with disjoint domain.

Figure 3 shows how the conventional language is extended to allow customizable composition operators.

As already mentioned, this is achieved as follows: *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). An example of meta-expression is

```
new C().m({ int m(){ return 2;} })
```

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).

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). Note that in rule (EXTENDS-2) in Figure 3 the fact that the left argument must be a value formally expresses the fact that evaluation goes from left to right.

We consider now the issue of soundness. Compile-time execution can: (1) not terminate; (2) get stuck; (3) reduce to a program where the right-hand-side of some class declaration is a value different from a class expression; (4) reduce to a program where some class declaration is ill-typed; (5) reduce to a well-typed program.

Indeed, there is no way to have *both* terminating metaprograms and a fully metacircular approach (over a Turing complete base-language) because, to guarantee the termination, one has to restrict either the resources used by metaprograms or the metalanguage itself. Pragmatically, users may want to enforce customizable limits on the resources used by the compilation process, for instance, CPU time usage and/or memory consumption, in order to avoid (what appear to be) non-terminating compilations.

The existence of similar limits is taken as a given in C++ [7], where the standard suggests[12] *minimum* quantities, for a number of limits, that any compiler should process; for instance, seventeen recursively nested template instantiations. Giving a minimum lower-bound makes sense when templates are used to describe parametric data structures, whereas, in our case, *maximum* quantities would be used to guarantee a terminating compilation. For this reason, these limits should not be values fixed once and for all, but rather compiler parameters. Indeed, only the authors of a piece of code can predict/decide whether the compilation process might need "more than usual" resources to (successfully) terminate or should be aborted because something has (probably) gone wrong.

To prevent (2)-(3)-(4), hence 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 4.

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`.

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.

[12] Those are guidelines only and do not determine compliance.

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

$$\text{(META-CHECK)}\ \frac{}{\begin{array}{l}cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C\ \texttt{=}\ e)\ p \longrightarrow \\ \qquad\qquad cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C\ \texttt{=}\ e{:}\texttt{code})\ \ p\end{array}}\ \Delta;\emptyset \vdash e{:}\texttt{code}$$

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

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

$$\text{(CHECK-ERROR)}\ \frac{}{cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p} \longrightarrow \texttt{error}}\ \begin{array}{l}cp' \neq \emptyset \\ \mathit{closed}(\Delta, cp') \text{ or } \tilde{p} = \emptyset \\ \nexists \Delta' \text{ s.t. } \Delta \vdash cp'{:}\Delta'\end{array}$$

Figure 4. Checked compile-time execution

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:

$$\tilde{p} ::= cp{:}\Delta\ cp'{:}\texttt{code}\ [\texttt{class}\ C\ \texttt{=}\ e{:}\texttt{code}]\ p \mid \texttt{error}$$

where square brackets denote optionality, and e is not of the form ce. Moreover, for any cp conventional program, $cp{:}\texttt{code}$ is the program obtained by annotating each class declaration by `code`, and, for any Δ s.t. $dom(cp) = dom(\Delta)$, $cp{:}\Delta$ 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 $cp{:}\Delta$ part comes first, then the $cp{:}\texttt{code}$ 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 later by the subject reduction property, that is, Theorem 2.2.

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 successfully 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, see the first example in the following.

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. 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.

We show now some examples illustrating how checked compile-time execution works.

First we give an example of successful compile-time execution. We abbreviate by B the base class `{ int one(){ return 1; } }`. The program

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

is reduced by (CHECK) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = {  int m() { return new E().one();} } : code
class E = new C().m()
```

is reduced by (META-CHECK) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = {  int m() { return new E().one();} } : code
class E = new C().m() : code
```

is reduced by (META-RED) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = {  int m() { return new E().one();} } : code
class E = { int one() { return 1; }  } : code
```

is reduced by (CHECK) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = {  int m() { return new E().one();} } :⟨∅,∅, int m() ⟩
class E = { int one() { return 1; }  } :⟨∅,∅, int one() ⟩
```

Compile-time execution checks that class `C` is well-typed. Note that it is not possible to check class `D` since it refers to class `E` that has no associated class expression yet. Hence, expression `new C().m()` is checked to be of type `code`. At this point, reduction of this expression can take place, and finally the resulting class `D` is checked to be well-typed. Finally, also the class `D` is verified to be well-typed.

The second example shows a case when compile-time execution terminates with an `error`.

```
class C = { code m() { return B; } }: code
class D = new C().k()
```

is reduced by (CHECK) to

```
 class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
 class D = new C().k()
```

is reduced by (META-CHECK-ERROR) to `error`.

Compile-time execution checks that class `C` is well-typed, and then checks whether the expression `new C().k()` is of type `code`. This is not the case, since class `C` has no methods named `k`. Moreover, because no standard typechecking steps are possible, since there are no other classes, an `error` is raised.

In the last example we abbreviate by B the base class
`{ int one(){ return new C().k(); } }`.

```
class C = { code m() { return B; } } : code
class D = new C().m()
```

is reduced by (CHECK) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = new C().m()
```

is reduced by (META-CHECK) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = new C().m() : code
```

is reduced by (META-RED) to

```
class C = { code m() { return B; } } :⟨∅,∅, code m() ⟩
class D = { int one() { return new C().k(); } } : code
```

is reduced by (CHECK-ERROR) to `error`.

Compile-time execution checks that class `C` is well-typed, then checks that the expression `new C().m()` is of type `code`, then reduces this expression. Finally, the check that the resulting class `D` is well-typed fails since class `C` has no methods named `k`.

This example also illustrates that standard typechecking of class expressions only takes place when they are associated to a class name in the class table. For instance, the fact that base class B is ill-typed is known from the beginning, but is only detected when B is associated to `D`. This choice allows for more expressive power, at the cost of a later error detection. In further work we will investigate smarter strategies allowing one to discover some inconsistencies earlier, for instance using type constraints as in [1].

In order to state our soundness result, we define a judgment $\vdash \tilde{p}\ \ \textsc{ok}$ which states that annotations in $\tilde{p}$ are correct.

$$\text{(OKERROR)}\ \frac{}{\vdash \texttt{error}\quad \text{OK}} \qquad \text{(OK1)}\ \frac{\emptyset \vdash cp{:}\Delta}{\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ p\quad \text{OK}}$$

$$\text{(OK2)}\ \frac{\begin{array}{c}\emptyset \vdash cp{:}\Delta \\ \Delta;\emptyset \vdash e{:}\texttt{code}\end{array}}{\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C\ \texttt{=}\ e{:}\texttt{code})\ p\quad \text{OK}}\ e \neq ce$$

Soundness is formally expressed by the usual progress and subject reduction properties. Proofs are in the Appendix.

Theorem 2.1 (Progress) *If* $\vdash \tilde{p}\quad$ OK*, then either* $\tilde{p} \longrightarrow \tilde{p}'$ *or* $\tilde{p} = \mathit{error}$ *or* $\tilde{p}$ *is of the form* $cp{:}\Delta$.

Theorem 2.2 (Subject reduction) *If* $\vdash \tilde{p}\quad$ OK *and* $\tilde{p} \longrightarrow \tilde{p}'$ *then* $\vdash \tilde{p}'\quad$ OK.

3 Conclusion

We have presented a framework for extending a Java-like language (that is, a class-based statically typed language with nominal types) with class composition operators blended into conventional expressions, thus using meta-programming as a flexible tool for composing software. Compile-time execution reduces extended class tables to conventional ones, by evaluating meta-expressions. Safety is ensured by a lightweight approach, where conventional typechecking of class expressions only takes place when they appear as class definitions in the class table.

An important advantage of this lightweight approach is that it is *modular*, in the sense that it can be applied on top of an existing Java-like language. In particular, checked compile-time execution is defined on top of typechecking and reduction relations of the underlying Java-like language, as formally shown in Figure 4. Correspondingly, an implementation could basically consist in [13] an algorithmic version of the rules in Figure 4 where (META-RED) steps and (META-CHECK)-(CHECK) steps invoke the JVM and the Java compiler, respectively.

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 [17], Prolog [15] and OpenJava [18] are meta-circular languages, while C++ [7], D [4], Meta-trait-Java [13] and MorphJ [5] use a specialized meta-language. [14] 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. The work presented in this paper lies in the area of meta-circular compile-time execution.

[13] Besides a parser for the extended language.

[14] The latest version of D seems to include a limited form of meta-circular compilation.

Among the above mentioned approaches, [18] is the one showing more similarities with ours. In OpenJava, programmers can add new language constructs on top of Java, and define the semantics of these new constructs by writing *meta-classes*, that is, particular Java classes which instruct the OpenJava compiler on how to perform a type-driven translation into standard Java. These meta-classes use the reflection-based *Meta Object Protocol (MOP)* to manipulate the source code. In the same way it is even possible to change the semantics of standard Java language constructs. A similar capability of specifying within the code instructions for contextual compilation has been recently provided in Java 6 by annotations.

However, this approach, besides being lower-level, has a very different goal w.r.t. ours, that is, to make easy for programmers to extend and possibly change the behaviour of an existing language, in rather arbitrary ways. In our case, instead, syntax and semantics of both the underlying language and the language for composing classes are fixed. The programmer is only allowed to define its own derived composition operators by using the whole expressive power of the underlying language. Note also that both approaches produce standard Java code; however, in our case this code is obtained by an algorithm which interleaves standard Java compilation and execution steps, rather than by a unique preprocessing step.

We conclude this summary of related work by a comparison with dynamically typed languages, which very often allow some sort of meta-circular facility, typically by offering an *eval* function. Such a function allows to run arbitrary code, represented by an input string. A more sophisticated approach is supported by Groovy[15], a dynamic language, targeting the JVM, that explicitly supports compile-time meta-programming via *AST (Abstract Syntax Tree) transformations*. That is, the Groovy compiler allows to specify *custom AST visitors*, which can arbitrarily modify the AST before it is turned into bytecode.

One pillar of our approach is the ability of tracing any type/composition problem back to the specific part of the source code that is responsible for it. This ability is the outcome of the following design choices: every class name is known from the start and every transformation (for instance, code merging or member renaming) is caused by the use of an explicit operator that can be found in the original source code. We do not consider these limitations as drawbacks since the “debuggability” of the resulting code is an utmost priority for us. The approach used by Grovvy, and in general by languages that expose the programmers to their internal AST, allows arbitrary code transformations and this suggests that it would be rather difficult, if possible at all, to trace back a problem found in the final AST. The attitude of trading off “debuggability” for flexibility sounds strikingly similar to the choice of a dynamically typed language versus a strongly typed one; in this sense, both choices seems perfectly consistent, and somewhat expected, in their respective languages, while they would probably make little sense in the other context. Evaluating other choices, in the wide spectrum from the total flexibility offered by an uncontrolled approach à-la Groovy to our restricted approach, is an interesting topic of future work.

[15] http://groovy.codehaus.org/

In this paper, we have illustrated our approach on a minimal class composition language, to be able to analyze in isolation the safety issue. Further work will be carried out in two complementary directions. On the one hand, we will design a richer composition language suitable for our aims, likely a subset/variant of Featherweight Jigsaw [11,10,9]. On the other hand, we will study alternative approaches to guarantee safety, ranging from a fully static analysis based on sophisticated types, as in MetaML [17], to intermediate solutions still including dynamic checks, but allowing earlier error detection. Moreover, to test the applicability of our proposal, we will develop a prototype which exploits our ideas by extending a real language such as Java.

Acknowledgement

We warmly thank the anonymous referees for their valuable help in improving the paper.

References

[1] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2005*. ACM Press, January 2005.

[2] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Gluck, David Vandevoorde, and Todd Veldhuizen. *Generative programming and active libraries (extended abstract)*, pages 25–39. Number 1766 in Lecture Notes in Computer Science. Springer, 2000.

[3] 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.

[4] Digital Mars. D programming language, 2007. http://www.digitalmars.com/.

[5] 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.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.

[7] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, 2003.

[8] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Customizable composition operators for Java-like classes (extended abstract). In *ICTCS'09 - Italian Conf. on Theoretical Computer Science*, 2009.

[9] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw — replacing inheritance by composition in Java-like languages. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, August 2009. Submitted for journal publication.

[10] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight Jigsaw - a minimal core calculus for modular composition of classes. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, number 5653 in Lecture Notes in Computer Science. Springer, 2009.

[11] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *FOOL'09 - Intl. Workshop on Foundations of Object Oriented Languages*, 2009.

[12] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[13] 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.

[14] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks.* PhD thesis, University of Bern, February 2005.

[15] Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques.* The MIT Press, April 1994.

[16] Bjarne Stroustrup. *The C++ Programming Language.* Reading. Addison-Wesley, special edition, 2000.

[17] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[18] 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.

A Proofs

Theorem A.1 (Progress) *If* $\vdash \tilde{p}\ \ \text{OK}$*, then either* $\tilde{p} \longrightarrow \tilde{p}'$ *or* $\tilde{p} = \texttt{error}$ *or* $\tilde{p}$ *is of the form* $cp{:}\Delta$*.*

Proof By case analysis on the definition of $\vdash \tilde{p}\ \ \text{OK}$.

(okError) Trivial.

(ok1) We have $\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ p\ \ \text{OK}$ and $\emptyset \vdash cp{:}\Delta$. We distinguish two subcases: either there exists a non-empty cp'' such that $cp'' \subseteq cp'$ and $closed(\Delta, cp'')$ or not.

cp'' **exists**

In this case,

- if $\Delta \vdash cp''{:}\Delta'$ for some Δ', then we can apply rule (CHECK),
- otherwise we can apply rule (CHECK-ERROR).

cp'' **not exists**

In this case,

- if p is empty and $cp'{:}\texttt{code}$ is not empty, then we can apply rule (CHECK-ERROR),
- if p is empty and $cp'{:}\texttt{code}$ is empty, then the program is of the form $cp{:}\Delta$,
- if p is not empty, then it is of the form $(C = e)\ p'$. In this case, if $\Delta; \emptyset \vdash e{:}\texttt{code}$, then we can apply rule (META-CHECK), otherwise rule (META-CHECK-ERROR).

(ok2) We have $\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C\ \texttt{=}\ e{:}\texttt{code})\ p\ \ \text{OK}$, with e not of the form ce, and $\emptyset \vdash cp{:}\Delta$, $\Delta; \emptyset \vdash e{:}\texttt{code}$. From these last two judgments and the fact that e is not a value, by the progress property of the conventional language we know that $e \xrightarrow{cp} e'$, hence we can apply rule (META-RED).

□

Lemma A.2 (Weakening) $\Delta; \Gamma \vdash ce{:}CT$ *implies* $\Delta, \Delta'; \Gamma \vdash ce{:}CT$*.*

Lemma A.3 *If* $\emptyset \vdash cp{:}\Delta$ *and* $\Delta \vdash cp'{:}\Delta'$ *then* $\emptyset \vdash cp, cp'{:}\Delta, \Delta'$*.*

Proof Since $\Delta \vdash cp'{:}\Delta'$ has been deduced by rule (PROGRAM-T), we have

(i) for each $C = ce$ in cp', $\Delta, \Delta' \vdash ce{:}\Delta'(C)$,

(ii) $\Delta, \Delta' \vdash \_ < \_$ is acyclic by side condition.

Analogously, since $\emptyset \vdash cp{:}\Delta$ holds, we have that, for each $C = ce$ in cp, $\Delta \vdash ce{:}\Delta(C)$. Hence, by Lemma A.2, $\Delta, \Delta' \vdash ce{:}\Delta(C)$, and we can apply (PROGRAM-T) getting the thesis. □

Theorem A.4 (Subject reduction) *If* $\vdash \tilde{p}\ \ \textsc{ok}$ *and* $\tilde{p} \longrightarrow \tilde{p}'$ *then* $\vdash \tilde{p}'\ \ \textsc{ok}$.

Proof By case analysis on the definition of $\tilde{p} \longrightarrow \tilde{p}'$.

(meta-red) We have

(i) $\tilde{p} \equiv cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e{:}\texttt{code})\ p \longrightarrow \tilde{p}' \equiv cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e'{:}\texttt{code})\ p$,
(ii) $e \xrightarrow{cp} e'$,
(iii) $\emptyset \vdash cp{:}\Delta$ and $\Delta; \emptyset \vdash e{:}\texttt{code}$, since $\vdash \tilde{p}\ \ \textsc{ok}$ holds.
From (ii) and (iii), by the subject reduction property of the conventional language, we get that $\Delta; \emptyset \vdash e'{:}\texttt{code}$. Hence, we can apply $(\textsc{ok}2)$ with this premise and get $\vdash \tilde{p}'\ \ \textsc{ok}$.

(meta-check) We have

(i) $\tilde{p} \equiv cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e)\ p \longrightarrow \tilde{p}' \equiv cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e{:}\texttt{code})\ \ p$.
(ii) $\Delta; \emptyset \vdash e{:}\texttt{code}$ by side condition.
(iii) $\emptyset \vdash cp{:}\Delta$ since $\vdash \tilde{p}\ \ \textsc{ok}$ holds.
Hence, we can apply $(\textsc{ok}2)$ with premises (ii) and (iii) and get $\vdash \tilde{p}'\ \ \textsc{ok}$.

(meta-check-error) We have $cp{:}\Delta\ cp'{:}\texttt{code}\ (\texttt{class}\ C = e)\ p \longrightarrow \texttt{error}$, hence we get the thesis by rule $(\textsc{okError})$.

(check) We have

(i) $cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p} \longrightarrow cp{:}\Delta\ cp'{:}\Delta'\ \tilde{p}$,
(ii) $\Delta \vdash cp'{:}\Delta'$ by side condition,
(iii) $\emptyset \vdash cp{:}\Delta$, since $\vdash cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p}\ \ \textsc{ok}$ holds,
From (ii) and (ii) we get $\emptyset \vdash cp, cp'{:}\Delta, \Delta'$ by Lemma A.3. Hence, we an apply $(\textsc{ok}1)$ with this premise and get $\vdash cp{:}\Delta\ cp'{:}\Delta'\ \tilde{p}\ \ \textsc{ok}$.

(check-error) We have $cp{:}\Delta\ cp'{:}\texttt{code}\ \tilde{p} \longrightarrow \texttt{error}$, hence we get the thesis by rule $(\textsc{okError})$.

□