

METAFJIG

A meta-circular composition language for Java-like classes

Marco Servetto Elena Zucca

DISI, Università di Genova
{servetto,zucca}@disi.unige.it

Abstract

We propose a Java-like language where class definitions are first class values and new classes can be derived from existing ones by exploiting the full power of the language itself, used on top of a small set of primitive composition operators, instead of using a fixed mechanism like inheritance.

Hence, compilation requires to perform (meta-)reduction steps, by a process that we call *compile-time execution*. This approach differs from meta-programming techniques available in mainstream languages since it is meta-circular, hence programmers are not required to learn new syntax and idioms.

Compile-time execution is guaranteed to be sound (not to get stuck) by a lightweight technique, where class composition errors are detected dynamically, and conventional typing errors are detected by interleaving typechecking with meta-reduction steps. This allows for a modular approach, that is, compile-time execution is defined, and can be implemented, on top of typechecking and execution of the underlying language. Moreover, programmers can handle errors due to composition operators.

Besides soundness, our technique ensures an additional important property called *meta-level* soundness, that is, typing errors never originate from (meta-)code in already compiled programs.

Categories and Subject Descriptors D.3 Programming Languages [Language Constructs and Features]

Keywords Java, meta-programming, module composition

1. Introduction

The recognized need of going beyond inheritance has led, in the last years, to a variety of proposals for improving the flexibility and expressivity of object-oriented programming, such as, e.g., mixin classes [4, 16], virtual classes [12, 13], generic classes as in Java 5, mixin modules [5, 14] and other proposals for adding a module/component level [1, 25], traits [11, 15, 24, 27].

All these proposals share a common limitation: users are provided with a fixed set of composition mechanisms.

A natural way to overcome this limitation is to allow programmers to write (meta-)code that can be used to generate customized code. However, meta-programming techniques provided in main-

stream Java-like languages,¹ such as *template meta-programming* in C++, can be very difficult to understand, since their syntax and idioms are esoteric compared to conventional programming, and well-formedness of generated code can only be checked “a posteriori”, making the whole process hard to debug.

In this paper, instead, we propose a *limited* meta-programming approach which is *meta-circular*² and *sound*. That is, respectively:

- Class definitions³ are first-class values, hence new classes can be derived from existing, rather than by a fixed mechanism like inheritance, by exploiting the full power of the language on top of a small set of primitive composition operators, as in [7, 8, 21]. Hence, compilation of a class table requires to perform (meta-)reduction steps, by a process that we call *compile-time execution*.
- Well-formedness of generated code is checked during compile-time execution itself, making the process easy to debug.

Here, rather than guaranteeing soundness employing (only) static checks, that would require a sophisticated type system with structural types for meta-expressions, we propose a lightweight solution, called *checked compile-time execution*, where typechecking at the conventional level is the standard one. However, meta-reduction steps are always performed on typechecked code, hence are safe. Type checking at the level of class composition operators, instead, is dynamic, that is, performed during meta-reduction. The motivation is twofold: to allow a modular approach, that is, compile-time execution is defined, and can be implemented, on top of typechecking and execution of the underlying language; and to allow programmers to handle composition errors, e.g., by relying on the Java exception mechanism.

Besides soundness, our technique ensures an additional important property called *meta-level* soundness. Intuitively, typing errors found during compile-time execution never originate from library (meta-)code, that is, programs which have already been compiled. This is not granted by other approaches like C++ templates.

We formally describe our approach by defining a calculus, called METAFJIG, where class definitions are first-class values which can be combined by using four primitive operators, namely *sum*, *restrict*, *alias*, and *redirect*. The conventional (that is, excluding meta-level features) fragment of METAFJIG is a subset of FJIG [21, 22], a Java-like calculus in the spirit of Featherweight Java [18] (FJ for short), where inheritance has been generalized to the much more flexible notion originally proposed in Bracha’s Jigsaw

¹By Java-like languages we mean object-oriented class-based languages with nominal types, such as Java, C++ and C#.

²Meta-circular languages are also called homogeneous, see, e.g., [29].

³But not other kinds of language terms: in this sense the meta-programming approach is limited.

framework [8]. Here, for sake of simplicity, we have omitted some FJIG features which are not relevant for the present work.

To show the effectiveness of our technique, we have implemented a prototype compiler, which can be downloaded (along with its sources and some examples) at:

<http://www.disi.unige.it/person/ServettoM/MetaFJig/>.

This prototype compiler supports a superset of the language used in the formal description, including primitive types `int` and `boolean`, predefined classes `Object`, `RuntimeException` and `String`, arrays, void methods, statements `if`, `while`, `try-catch`, `for-each`, variable declaration and assignment.

The conventional and meta-level features of METAFJIG are informally presented in Section 2.1 and Section 2.2, respectively, whereas Section 2.3 shows some applications. The examples are written in the extended syntax supported by the prototype compiler, hence they can all be tested. Section 3 contains the definition of the calculus. Section 4 formally defines checked compile-time execution and states its properties. Section 5 describes how the prototype compiler works, notably explaining how it is built on top of Java compiler and Java Virtual Machine. Finally, Section 6 outlines related and further work.

A very preliminary presentation of the ideas in this paper has been given in [20, 23].

2. An informal introduction

2.1 Primitive composition operators

In METAFJIG, a class declaration associates an expression of primitive type `class` with a class name. The simplest form of such an expression is a *class constant*, which is similar to a Java class body and contains, besides other components explained later on, abstract or defined member (field or method) declarations.

The following example shows two class declarations where class constants are directly associated to class names.

```
class A = {
  abstract int m1();
  int m2() { return m1() + 1; }
}
class B = {
  abstract int m2();
  int m1() { return 1 + m2(); }
}
```

These two classes are abstract (hence cannot be instantiated).

Compound expressions of type `class` can be constructed using *primitive composition operators*. For instance, a concrete class can be obtained from those above by applying the *sum* operator as follows:

```
class Sum = A [+] B
```

This declaration is reduced to the following:

```
class Sum = {
  int m1() { return 1 + m2(); }
  int m2() { return m1() + 1; }
}
```

Conflicting definitions for the same member are not permitted, whereas abstract members with the same name are shared. Besides *sum*, METAFJIG provides other three primitive composition operators: *restrict*, *alias*, and *redirect*, which take as arguments a class and one or two (member) names, that is, expressions of primitive type `name`. The *restrict* operator removes a definition, making the corresponding member abstract. The *alias* operator adds a definition for an either abstract or new member, duplicating that of an existing member. The *redirect* operator replaces all the internal references (in method bodies or field initialization expressions) to an

abstract member name by a different name, and removes its declaration⁴. For instance,

```
class Restrict = Sum[restrict $m1]
class Alias = A[alias $m2 to $m1]
class Redirect = A[redirect $m1 to $m2]
```

is reduced to

```
class Restrict = {
  abstract int m1();
  int m2() { return m1() + 1; }
}
class Alias = {
  int m1() { return m1() + 1; }
  int m2() { return m1() + 1; }
}
class Redirect = {
  int m2() { return m2() + 1; }
}
```

The notation $\$n$ is used for the constant of type `name` corresponding to name n .

These four primitive operators are a minimal, yet very expressive set. *Sum*, *restrict*⁵ and *alias* are very similar to the trait operators defined in [11], with the difference that they uniformly work on fields as well, as shown below. Other trait-based languages [7, 26] also include a *rename* operator. We prefer to have the *redirect* operator since it is conceptually more primitive, and it allows to express many different forms of renaming, see the examples in Section 2.2.⁶

In METAFJIG, as in FJIG, the modifier *abstract* applies to fields as well, as shown by the following example which also illustrates how constructors work.

```
class A1 = {
  abstract int f1;
  int f2;
  constructor(int x) { f2 = x; }
  int m() { return f1 + f2; }
}
class B1 = {
  int f1;
  abstract int f2;
  constructor(int x) { f1 = x + 1; }
} [+] A1
```

A class constant defines one⁷ constructor which specifies a sequence of parameters and a sequence of initialization expressions, one for each non-abstract field. We assume a default constructor with no parameters for classes having no defined fields. In order to be composed by the *sum* operator, two classes should provide a constructor with the same parameter list. The effect is that the resulting class provides a constructor with the same parameter list, that executes both the original constructors.

Classes composed by *sum* can share the same field, provided it is abstract in all except (at most) one. Note that this corresponds to *sharing* fields as in, e.g., [6]; however, in our framework we do not need an ad-hoc notion.

⁴ Unless it is a member of a *supertype*, see later on.

⁵ *Restrict* is called *exclude* in trait literature; here we prefer to stick to the original name in Jigsaw [8].

⁶ In METAFJIG we have omitted some FJIG features not relevant for the present work. Notably, in Jigsaw and FJIG defined members can be *virtual*, *frozen* or *local*, whereas here they are all implicitly virtual. As a consequence, METAFJIG does not include the *freeze* primitive operator which allows to express, e.g., member hiding. Moreover, the *reduct* operator of FJIG, handling maps from names into names, has been replaced by three operators which handle single names (*restrict*, *alias* and *redirect*). They provide the same expressive power and are more convenient for the meta-level.

⁷ As in FJ and differently from Java, overloading is not allowed.

METAJIG keeps the Java nominal approach, that is, types are class names. However, desired subtyping relations must be explicitly written by the programmer, by declaring a set $C_1 \dots C_n$ of *supertypes*, introduced by the keyword `implements`, in a class constant. This is also used to typecheck occurrences of `this` inside method bodies⁸, as illustrated by the example below.

```
class C = {
  abstract int m1();
  abstract int m2();
}
class D = implements C {
  abstract int m1();
  int m2() { return 1 + m1(); }
  C m() { return this; }
}
```

The type system checks, for each C_i , that the subtyping relation can be safely assumed, that is, members of C_i are members of the class constant as well⁹. We assume as default supertype the predefined class `Object` with no members.

To conclude this overview of the conventional features of METAJIG, note that $m(\dots)$ is *not* a shortcut for `this.m(\dots)`, but a different form of invocation, called *internal*, whereas `this.m(\dots)` is a special case of client invocation where the receiver is the current object. Only the former is affected by composition operators. For instance, with `C` as above:

```
class D2 = implements C{
  abstract int m1();
  abstract int m2();
  int m3(D2 x){return m1() + x.m1()+ this.m1();}
}[$m1 redirect $m2]
```

is reduced to

```
class D2 = implements C{
  abstract int m1();
  abstract int m2();
  int m3(D2 x){return m2() + x.m1()+ this.m1();}
}
```

However, this difference is no longer relevant when all class definitions have been reduced to class constants.

2.2 Meta-level features

Since `class` is a primitive type of the language, a class definition can be the result of a method. For instance, in the following meta-program¹⁰

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

method `m` returns a value of type `class`. This value is a class constant declaring the method `one`. Class `D` is defined by an expression that has to be evaluated in order to obtain the corresponding class constant. In this example, the definition 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; } }
```

⁸ See rule (THIS-T) in Figure 6.

⁹ Formally, that the type of `this` is subtype of the structural type of C_i , see rule (CLASS-T) in Figure 5.

¹⁰ A METAJIG *meta-program* is an arbitrary sequence of class declarations, whereas a *program* is a sequence where all right-hand-sides are class constants.

One very basic use of this mechanism allows to obtain conditional compilation. For instance:

```
class C = {
  class m() {
    if (DEBUG) return /*debug version*/;
    else return /*release version*/;
  }
}
```

The following method:

```
class mixinFoo(class parent) {
  return { /* ... */ } [+] parent;
}
```

behaves like a *mixin* class, extending in some way a parent class passed as argument¹¹.

The class to be used as parent could be constructed, having a class `ClassList` implementing lists of classes, by chaining an arbitrary number of classes:

```
class chain(ClassList parents){
  if (parents.isEmpty()) return {};
  else return parents.head() [+]
    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.

We show now how to derive other useful composition operators from the primitive ones. To this end, we use an additional primitive operator *members* which returns the array of member names of a class, omitted in the formalization in Section 3 since it poses no significant new technical problems. This operator allows to emulate a type-driven translation. Moreover, we use the following class

```
class IsDefined = {
  boolean apply(class c, name n) {
    try{c[restrict n];
      return true;}
    catch(RestrictException e){return false;}
  }
}
```

which defines an operator¹² which checks whether a member is defined (that is, declared and non abstract) in a class. This operator can be derived from the `restrict` operator, which throws an exception when invoked on a non defined member.

The following class defines the `override`¹³ operator as the composition of `restrict` and symmetric sum [8].

```
class Override = {
  class apply(class h, class p){
    for(name n:h[members])
      if (new IsDefined().apply(h,n) &&
          new IsDefined().apply(p,n))
        p=p[restrict n];
    try{return h [+] p;}
    catch(SumException se){
      throw new OverrideException("...");
    }
  }
}
class OverrideException=
  implements CompositionException{...}
```

¹¹ Instead of sum, we should more appropriately use the derived `override` operator described in the sequel.

¹² The operators in this sequence of examples should better be implemented as static methods, but these are not supported yet by our prototype.

¹³ Corresponding to Java *extends*.

For all members which are defined both in “heir” *h* and in “parent” *p*, the definition in *p* has to be removed. An exception can still be raised if some member is declared in both with different types.¹⁴ The predefined `CompositionException` should be a supertype of all the exceptions modeling composition errors.

The following class defines a rename operator as, e.g., in [7].

```
class Rename = {
  class apply(class c, name o, name n){
    if(!isIn(o,c[members]))
      throw new RenameException("...");
    if(o==n)return c;
    if(isIn(n,c[members]))
      throw new RenameException("...");
    if(new IsDefined.apply(o,c))
      c=c[alias o to n][restrict o];
    return c[o redirect n];
  }
  boolean isIn(name n,name[] ns){...}
}
```

The operator can be applied only if the old name *o* is declared in *c* and the new name *n* is not. If *o* is defined, then its definition is moved to *n*. Finally, the occurrences of (now certainly abstract) *o* as internal references are replaced by *n*.

The following classes define two less standard operators.

```
class StrongAlias = {
  class apply(class c, name o, name n){
    try{return c[alias o to n][restrict o]
      [n redirect o][alias n to o];}
    catch(CompositionException ce){
      throw new StrongAliasException("...");
    }
  }
}
```

```
class Unbind = {
  class apply(class c, name o, name n){
    try{return StrongAlias.apply(c,o,n)
      [restrict n];}
    catch(CompositionException ce){
      throw new UnbindError("...");
    }
  }
}
```

The former is a variant of `alias` which also replaces internal references¹⁵. This operator works in four steps: duplicates the definition of *o* as *n*, removes the definition of *o*, redirects the (now abstract) name *o* to *n*, and finally restores the original definition of *o*.

The latter replaces internal references to a defined member *o* by a new name *n*. This effect can be obtained by first applying `StrongAlias`, then removing the definition of *n*. This operator has been introduced in [5] to deal with unanticipated code modification due to poor design. The converse operator, which binds internal references to an existing defined member, is a special case of `redirect`.

2.3 Real world examples

The following example is a graphical library that adapts itself with respect to its execution environment, without requiring any extra-linguistic mechanisms:¹⁶

```
class GraphicalLibrary = {
  class produceLibrary() {
    class result = BaseGraphicalLibrary;
    String producer =
      System.getProperty("sys.vcard.brand");
    if (producer.equals("NVIDIA"))
      result = NVIDIASupport [+] result;
    else if (producer.equals("ATI"))
      result = ATISupport [+] result;
    else
      throw new UnsupportedOperationException(
        "No compatible hardware found");
    if (System.getProperty("os.name")
      .contains("Windows"))
      result = CygwinAdapter.adapt(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* [9], that is, libraries that interact dynamically with the compiler, providing better services, as meaningful error messages, platform-specific optimizations and so on.

Finally, one simple and interesting application is in managing at compile-time external applications like databases. For instance, the following class `DBRecord` provides the method `create` that, taken a table name, produces a class which mimics the structure of the table.

```
class DBRecord = {
  StringClassMap map=...
  //{"int"->{int n;constructor(){n=0;}},
  //{"String"->{String n;constructor(){n=""}}},
  //...};
  class create(String table){
    TableStructure structure=
      DB.getTableStructure(table);
    class result={};
    for(Column c:structure.getColumns())
      result=result [+]
        Rename.apply(map.get(c.type),$n,c.name);
    return result;
  }
}
```

The `create` method takes the name of a table, and for all the `Column c` of such table adds to the class stored in `result` a field with the type and name of *c*.

Many applications rely on the fact that the shape of some database table is known and immutable, but with conventional approaches the programmer needs to duplicate the structure of the external table in the code, and checks on the table shape are performed only at runtime. With our approach, instead, a class defined by

```
class Foo = DBRecord.create("Foo")
```

will have the shape of the table "Foo" in the DB. That is, the following code:

```
int m(Foo x){return x.bar;}
```

¹⁴ See rule (SUM-ERROR) in Figure 2.

¹⁵ A similar operator is defined in [24], but only affects internal references appearing in the body of the method itself.

¹⁶ 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.

will be successfully compiled only if the table "Foo" has a field `bar` of type `int`. That is, checks on the table shape are performed at compile-time.¹⁷

The two examples above also show that our approach allows, in a statically typed setting, an expressive power which is typical of dynamically typed languages.

3. The language

3.1 Syntax and reduction rules

In Figure 1 we give the syntax of METAFJIG. We use the bar notation for sequences, e.g., \bar{d} is a sequence of declarations d . We assume infinite sets of *class names* C , (*member*) *names* n , and *variables* x .

mp	$::= \overline{\text{class } C=e}$	meta-program
p	$::= \overline{\text{class } C=c}$	program
c	$::= \text{implements } \overline{C\{k \bar{d}\}}$	class constant
k	$::= kh\{\bar{fe}\}$	constructor
kh	$::= \text{constructor}(\overline{T x})$	constructor header
fe	$::= f=e;$	field expression
d	$::= ad \mid fd \mid md$	declaration
ad	$::= \text{abstract } fd \mid \text{abstract } mh;$	abstract declaration
fd	$::= T f;$	field definition
mh	$::= T m(\overline{T x})$	method header
md	$::= mh\{\text{return } e;\}$	method definition
T	$::= C \mid \text{name} \mid \text{class}$	type
e	$::= x \mid f \mid m(\bar{e})$ $\mid e.f \mid e.m(\bar{e}) \mid \text{new } C(\bar{e})$ $\mid C(\bar{fe})$ $\mid n \mid C \mid c \mid e_1[+]e_2$ $\mid e_1[\text{alias } e_3 \text{ to } e_2]$ $\mid e_1[\text{restrict } e_2]$ $\mid e_1[\text{redirect } e_2 \text{ to } e_3]$	expression (conventional) expression (pre-object) expression (meta-level)
f	$::= n$	field name
m	$::= n$	method name
v	$::= C(\bar{fv}) \mid n \mid c$	value
fv	$::= f=v;$	field value

Figure 1. Syntax

A meta-program is a sequence of class declarations, where an expression is associated to a class name. A program is a meta-program where all these expressions are class constants.

A class constant declares a sequence of *supertypes*, a constructor, and a sequence of (member) declarations, which can be either abstract or non abstract (member definitions). The first component means that the type of `this` must be a subtype of (the structural type of) C for each class name C in the sequence. This information is needed to typecheck occurrences of `this` in method bodies, see [7, 22]. There is no overloading, hence a class has only one constructor. However, differently from FJ, where this unique constructor has a canonical form, there is no a priori relation among the parameter list and the constructor body which is a sequence of *field expressions* associating (initialization) expressions to field names. Members can be fields or methods, and member declarations are in the style of FJ.

¹⁷ However, since the table shape can change after compilation, the code that reads the table will perform integrity checks at runtime.

Sequences of class declarations, `this` supertypes, declarations, and field expressions are considered as sets, that is, order and repetitions are immaterial. Moreover, in a well-formed (meta-)program, a class name cannot be declared twice, that is, a (meta-)program is a map from class names to expressions, hence we can safely use the notation $mp(C)$. Analogously, no member can be declared twice in a class constant. This implies that, differently from Java, there is no method overloading, and there is no overloading between field and method names. However, for better readability, we will use the metavariable f when a name is used for a field, m for a method. A parameter name cannot be declared twice in a constructor or method header. Finally, a field name cannot appear twice in a sequence of field expressions, hence we can safely use the notation $\bar{fe}(f)$, and there is exactly a field expression in the constructor for each non abstract field.

Types are class names and the primitive types `name` of names and `class` of classes.

Expressions in method bodies include conventional constructs, pre-objects, and meta-expressions.

Conventional constructs are similar to those of FJ. We omit cast for simplicity since it is not relevant for our technical treatment; moreover, we distinguish between *internal* field accesses and method invocations, which have the current object `this` as implicit receiver, and *external* field accesses and method invocations which have an explicit receiver. As explained at the end of Section 2.1, only internal references are affected by composition operators.

Pre-objects are runtime expressions which cannot be written in programmer's code, but are obtained by reducing a constructor invocation. Indeed, since the constructor has no canonical form, we need two different syntactic forms [21, 22], differently from FJ.

Meta-expressions are names, class names, class constants, or are constructed by the four operators *sum*, *reduct*, *alias*, and *redirect*. The behaviour of these operators will be described when explaining the corresponding reduction rules.

In the concrete syntax used by the prototype, we use the notation $\$n$ for the constant meta-expression denoting the name n , which needs to be distinguished from the conventional expression n , which is an internal field access expected to be bound to a field name existing in the current scope.

Values are objects, names and class constants. Objects are pre-objects where all field expressions are (recursively) values.

In Figure 2 we give the rules which define the reduction of an expression e in the context of a program p . We omit standard rules for contextual closure and error propagation.

We use the following notations, informally defined for brevity (formalization is straightforward):

- $mbody_p(C, m)$ gives parameters and body of method m in class C in p ;
- \hat{e} is the expression obtained from e by replacing internal references (that is, internal field accesses and method invocations) by the corresponding external versions with receiver `this`;
- $names(\bar{d})$, $abs(\bar{d})$ and $def(\bar{d})$ is the set of all, abstract and defined names declared in \bar{d} , respectively; $names(p, C) = names(\bar{d})$ if $p(C) = \text{implements } _ \{ _ \bar{d} \}$;
- if $n \in names(\bar{d})$, then $mtype(n, \bar{d})$ is its type (member types are defined in Figure 3);
- $\bar{d} \parallel \bar{d}'$ means that \bar{d} and \bar{d}' are *non conflicting*, that is, $def(\bar{d}) \cap def(\bar{d}') = \emptyset$, and *coherent*, that is, for all $n \in names(\bar{d}) \cap names(\bar{d}')$, $mtype(n, \bar{d}) = mtype(n, \bar{d}')$;
- $\bar{d}[\text{restrict } n]$ is obtained from \bar{d} by making abstract the non abstract declaration of n , if any, or removing the abstract declaration of n , if any; analogously, $\bar{fe}[\text{restrict } n]$ is obtained

$$\begin{array}{c}
\text{(CLIENT-FIELD)} \frac{\overline{C(\overline{fv})}.f \xrightarrow{p} v}}{\overline{fv}(f)=v} \quad \text{(CLIENT-INVK)} \frac{\overline{v.m(\overline{v})} \xrightarrow{p} \hat{e}[\overline{v}/\overline{x}][v/\text{this}]}{mbody_p(C, m)=\overline{x}; e} \\
\text{(OBJ-CREATION)} \frac{\overline{\text{new } C(\overline{v})} \xrightarrow{p} C(\overline{fe}[\overline{v}/\overline{x}])}{\begin{array}{l} p(C)=\text{implements_}\{\text{constructor}(_ \ x_1 \dots \ x_n)\{\overline{fe}\} \overline{d}\} \\ \text{abs}(\overline{d}) = \emptyset \\ \overline{x} = x_1 \dots \ x_n \end{array}} \quad \text{(C-NAME)} \frac{\overline{C} \xrightarrow{p} C}{p(C)=c} \\
\text{(SUM)} \frac{\overline{\text{implements } \overline{C}_1\{kh\{\overline{fe}_1\} \overline{d}_1\}[+]\text{implements } \overline{C}_2\{kh\{\overline{fe}_2\} \overline{d}_2\}} \xrightarrow{p} \overline{\text{implements } \overline{C}_1 \overline{C}_2\{kh\{\overline{fe}_1 \overline{fe}_2\} \overline{d}'_1 \overline{d}'_2\}}}{\begin{array}{l} \overline{d}_1 \parallel \overline{d}_2 \\ \overline{d}'_1 = \overline{d}_1[\text{restrict } def(\overline{d}_2)] \\ \overline{d}'_2 = \overline{d}_2[\text{restrict } def(\overline{d}_1)] \end{array}} \\
\text{(SUM-ERROR)} \frac{\overline{\text{implements } _ \{kh_1\{_ \} \overline{d}_1\}[+]\text{implements } _ \{kh_2\{_ \} \overline{d}_2\}} \xrightarrow{p} \text{error}}{kh_1 \neq kh_2 \text{ or } \overline{d}_1 \not\parallel \overline{d}_2} \\
\text{(RESTRICT)} \frac{\overline{\text{implements } \overline{C}\{kh\{\overline{fe}\} \overline{d}\}[\text{restrict } n]} \xrightarrow{p} \overline{\text{implements } \overline{C}\{kh\{\overline{fe}[\text{restrict } n]\} \overline{d}[\text{restrict } n]\}}}{n \in def(\overline{d})} \\
\text{(RESTRICT-ERROR)} \frac{\overline{\text{implements } _ \{ _ \} \overline{d}\}[\text{restrict } n]} \xrightarrow{p} \text{error}}{n \notin def(\overline{d})} \\
\text{(ALIAS)} \frac{\overline{\text{implements } \overline{C}\{kh\{\overline{fe}\} \overline{d}\}[\text{alias } n \text{ to } n']} \xrightarrow{p} \overline{\text{implements } \overline{C}\{kh\{\overline{fe}\} \overline{d}\}[\text{alias } n \text{ to } n']}}{\begin{array}{l} n \in def(\overline{d}) \\ n' \notin def(\overline{d}) \\ n' \in abs(\overline{d}) \text{ implies } mtype(n, \overline{d}) = mtype(n', \overline{d}) \end{array}} \\
\text{(ALIAS-ERROR)} \frac{\overline{\text{implements } _ \{ _ \} \overline{d}\}[\text{alias } n \text{ to } n']} \xrightarrow{p} \text{error}}{\begin{array}{l} n \notin def(\overline{d}) \text{ or } \\ n' \in def(\overline{d}) \text{ or } \\ mtype(n, \overline{d}) \neq mtype(n', \overline{d}) \end{array}} \\
\text{(REDIRECT)} \frac{\overline{\text{implements } \overline{C}\{k \overline{d}\}[\text{redirect } n \text{ to } n']} \xrightarrow{p} \overline{\text{implements } \overline{C}\{k \overline{d}'\}}}{\begin{array}{l} n \in abs(\overline{d}) \\ n' \in names(\overline{d}) \text{ implies } mtype(n, \overline{d}) = mtype(n', \overline{d}) \\ \overline{d}' = \begin{cases} \overline{d}[\text{redirect } n \text{ to } n'][\text{restrict } n] & \text{if } n \notin names(p, C) \text{ for all } C \in \overline{C}, \\ \overline{d}[\text{redirect } n \text{ to } n'] & \text{otherwise} \end{cases} \end{array}} \\
\text{(REDIRECT-ERROR)} \frac{\overline{\text{implements } \overline{C}\{ _ \} \overline{d}\}[\text{redirect } n \text{ to } n']} \xrightarrow{p} \text{error}}{n \notin abs(\overline{d}) \text{ or } mtype(n, \overline{d}) \neq mtype(n', \overline{d})}
\end{array}$$

Figure 2. Reduction rules

from \overline{fe} by removing the field expression for n , if any; these notations are generalized to set of names;

- $\overline{d}[\text{alias } n \text{ to } n']$ is obtained from \overline{d} by adding a definition for n' equal to that for n , and removing the abstract declaration for n' , if any; analogously, $\overline{fe}[\text{alias } n \text{ to } n']$ is obtained from \overline{fe} by adding a field expression for n' equal to that for n , if any;
- $\overline{d}[\text{redirect } n \text{ to } n']$ is obtained from \overline{d} by replacing internal references to n in method bodies by n' , and adding an abstract declaration for n' if $n' \notin names(\overline{d})$.

The first three rules define reduction of conventional constructs.

There are no reduction rules for variables (parameter names) and internal field accesses and method invocations since those appearing in a method body are replaced at invocation time¹⁸, see (CLIENT-INVK).

Rules (CLIENT-FIELD) and (CLIENT-INVK) are as in FJ, with the difference that, since there is no inheritance, method look-up (modeled by $mbody$) is trivial, and method body e is transformed in \hat{e} .

Rule (OBJECT-CREATION) is straightforward and reduces a constructor invocation into the pre-object obtained by replacing parameters by corresponding arguments in the constructor's body.

¹⁸ Indeed, as explained at the end of Section 2.1, the difference between, say, f and $\text{this}.f$ is only relevant for composition operators, notably redirect.

Note that only classes with no abstract members can be instantiated.

The following rules define reduction of meta-expressions, that is, expressions of type `class`.

Rule (C-NAME) reduces the name of a class declared in the program into the corresponding definition.

For each composition operator there are two rules, the latter corresponding to the case when the operator cannot be performed, hence an error is raised. In the prototype compiler, of course, a different predefined exception is thrown for each operator.

The sum operator merges two class constants, as shown in rule (SUM). The arguments must have the same constructor header¹⁹, non conflicting and coherent declarations, that is, no member can be defined in both and, if declared in both, it must have the same type. Then, the result has the constructor header of the arguments, the (disjoint) union of the field expressions, the (disjoint) union of the defined members, and the union of the declared members, which are abstract only if abstract in both arguments. Note that the side conditions ensure well-formedness of the result. If one of the side conditions does not hold, then rule (SUM-ERROR) is applied.

The restrict operator replaces the definition of n by the corresponding abstract declaration, as shown in rule (RESTRICT). If n is

¹⁹ In order to make equal the constructor header of two classes, FJG [21, 22] provides a *constructor wrapper* operator, omitted here for simplicity.

Δ	$::=$	$\overline{C}:CT$	class type environment
CT	$::=$	$\overline{C}; \overline{T}; ck; \Sigma$	class type
ck	$::=$	$\mathbf{a} \mid \neg \mathbf{a}$	class kind
Σ	$::=$	$n:MT$	signature
MT	$::=$	$T \mid \overline{T} \rightarrow T$	member type
Π	$::=$	$x:\overline{T}$	parameter type environment

Figure 3. Type environments

a field, its initialization expression is removed as well. If n is not defined, then rule (RESTRICT-ERROR) is applied.

The alias operator adds a definition for n' , by duplicating that existing for n . The name n' can be either new or abstract, in which case the definition replaces the previous abstract declaration, and the two must be coherent. If n is a field, then the initialization expression is duplicated as well. Note also that, if n is a method, then recursive internal references to n in its body are not affected when the body is duplicated for n' . If n is not defined, or n' is already defined, or its declaration is not coherent with the definition of n , then rule (ALIAS-ERROR) is applied.

The redirect operator replaces internal references to abstract member n in method bodies by n' . If n' was not a member yet, then a corresponding abstract declaration for n' is added. Otherwise, the declaration of n' must be coherent with that of n . The abstract declaration for n is removed, unless it is required by the assumption that, for all $C \in \overline{C}$, C can be safely assumed as supertype for `this`, that is, all members of C have a (coherent) declaration in \overline{d} as well. If n is not abstract, or the declaration of n is not coherent with that of n' , then rule (REDIRECT-ERROR) is applied.

3.2 Type system

Type environments are defined in Figure 3.

A class type environment is a sequence of associations from class names to class types, assumed to be a map. A class type is a 4-tuple, consisting of the supertypes (a sequence of class names), constructor type (the sequence of constructor parameter types), the class kind (abstract or non abstract) and the class signature. A signature is a sequence of associations from member names to member types, assumed to be a map. A field type is just a type, whereas a method type is a pair consisting of the sequence of parameter types and the result type. Finally, a parameter type environment is a sequence of associations from variables (parameter names) to types, assumed to be a map.

We distinguish two notions of typing for METAFJIG programs: (*standard*) *well-typedness* and *strong well-typedness*. The difference is illustrated by the following example. In the program

```
class D = {int foo(){return 0;}}
class C = {int m(){return new D().bar();}}
```

the class constant used as definition of `C` is ill-typed (there is no method `bar` in class `D`), hence the whole program is ill-typed. However, a similar program where the ill-typed class constant is only used as meta-expression, as the following:

```
class D = {int foo(){return 0;}}
class C = {class m(){return
  { int n(){return new D().bar();}}
}}
```

is considered well-typed, but not strongly well-typed.

We denote the two typing judgments by \vdash^0 and \vdash^* , respectively, to suggest that class constants must denote well-typed classes in the former case only when used as class definitions (that is, at level 0), in the latter case at any inner meta-level. Well-typedness is enough to guarantee (standard) soundness, see Theorem 1. Strong

$$\begin{array}{c}
\text{(CLASS-CONSTANT-T}^0\text{)} \frac{}{\Delta; _; _; _ \vdash^0 \text{implements } \overline{C}\{_ _ \}: \text{class}} \quad \overline{C} \subseteq \text{dom}(\Delta) \\
\text{(CLASS-CONSTANT-T}^*\text{)} \frac{\Delta \vdash^* c:CT}{\Delta; _; _; _ \vdash^* c: \text{class}}
\end{array}$$

Figure 4. The two typing rules for class constants

$$\begin{array}{c}
\text{(PROGRAM-T)} \frac{\Delta \Delta^p \vdash c_i \forall i \in 1..n}{\Delta \vdash p} \quad p = \text{class } C_1 = c_1 \dots \text{class } C_n = c_n \\
\text{(CLASS-T)} \frac{\Delta; \Sigma^{\overline{d}} \vdash k}{\Delta \vdash \text{implements } \overline{C}\{k \overline{d}\}} \quad \overline{d} = \overline{ad} \overline{fd} \overline{md} \quad \Delta \vdash \Sigma^{\overline{d}} \leq \text{sig}(\Delta, C) \forall C \in \overline{C} \\
\text{(METHOD-T)} \frac{\Delta; \Sigma; \Gamma; \overline{C} \vdash e: T'}{\Delta; \Sigma; \overline{C} \vdash T m(\overline{T} x) \{ \text{return } e; \}} \quad \begin{array}{l} \Gamma = x_1:T_1 \dots x_n:T_n \\ \overline{T} x = T_1 x_1 \dots T_n x_n \\ \Delta \vdash T' \leq T \end{array} \\
\text{(CONS-T)} \frac{\Delta; \emptyset; \Gamma; \emptyset \vdash e_i:T'_i \forall i \in 1..k}{\Delta; \Sigma \vdash \text{constructor}(\overline{T} x) \{ f_1=e_1 \dots f_k=e_k \}} \quad \begin{array}{l} \Gamma = x_1:T_1 \dots x_n:T_n \\ \overline{T} x = T_1 x_1 \dots T_n x_n \\ \Delta \vdash T'_i \leq \Sigma(f_i) \forall i \in 1..k \end{array}
\end{array}$$

Figure 5. Typing rules for programs and classes

well-typedness is needed to guarantee *meta-level soundness*, an additional important property discussed later on, see Theorem 7.

Formally, the only difference in the definition of the two judgments is in the rule for typing class constants, whose two versions are given in Figure 4.

In order to be a well-typed class meta-expression, the only condition a class constant must satisfy is that class names appearing as this supertypes actually exist, as shown in rule (CLASS-CONSTANT-T⁰). This condition is necessary to perform the check required in rule (REDIRECT), otherwise reduction could go stuck. Instead, rule (CLASS-CONSTANT-T^{*}) states that a class constant is a strongly well-typed meta-expression if it is well-typed as class.

The other rules are given in Figure 5 and Figure 6, where \vdash must be replaced by \vdash^0 and \vdash^* , respectively. We use the following notations:

- $\Sigma^{\overline{d}}$ is the signature extracted from \overline{d} , that is, $\Sigma^{\overline{d}}(n) = \text{mtype}(n, \overline{d})$;
- CT^c is the class type extracted from c , that is, $CT^c = \overline{C}; T_1 \dots T_n; ck; \Sigma^{\overline{d}}$ if $c = \text{implements } \overline{C}\{\text{constructor}(T_1 x_1 \dots T_n x_n)\{_ _ \} \overline{d}\}$, with $ck = \mathbf{a}$ if $\text{abs}(\overline{d}) \neq \emptyset$, $ck = \neg \mathbf{a}$ otherwise;
- Δ^p is the class type environment extracted from p , that is, $\Delta^p(C) = CT^c$ if $p(C) = c$;
- if $\Delta(C) = \overline{C}; \overline{T}; ck; \Sigma$, then $k\text{type}(\Delta, C) = \overline{T}$, $\text{sig}(\Delta, C) = \Sigma$, $\text{kind}(\Delta, C) = ck$;
- $\text{mtype}(\Delta, C, n) = \text{sig}(\Delta, C)(n)$.

Rules in Figure 5 define the typing judgment $\Delta \vdash p$, meaning that program p is well-typed w.r.t. the class type environment Δ , modeling external libraries, and the typing judgments for classes, method definitions and constructors.

The rules are straightforward. Note the side-condition in rule (CLASS-T) checking that the signature of the class is actually a

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

Figure 7. Compile-time execution

subtype of the signature of all `this` supertypes.²⁰ The (standard) definitions of the nominal subtyping relation $\Delta \vdash T \leq T'$ and of the structural subtyping relation $\Delta \vdash \Sigma \leq \Sigma'$ are given in the Appendix, Figure 11. Also note, comparing the form of typing judgment for expressions explained below with the premise of rule (CONS-T) that, as expected, field expressions can neither contain internal references ($\Sigma = \emptyset$), nor `this` ($C = \emptyset$).

Rules in Figure 6 define the typing judgment $\Delta; \Sigma; \Pi; \overline{C} \vdash e: T$, meaning that expression e has type T w.r.t. the class type environment Δ , needed to typecheck client field accesses and method invocations, and constructor invocations, the signature Σ , needed to typecheck internal field accesses and method invocations, the parameter type environment Π , needed to typecheck variables, and the sequence of class names \overline{C} , needed to typecheck `this` occurrences.

The rules are straightforward.

The type system is sound, as formally stated by the following standard theorem.

Theorem 1 (Soundness). *If $\emptyset \vdash p$, $\Delta^p; \emptyset; \emptyset; \emptyset \vdash e: T$, and $e \xrightarrow{p} e'$, then either e' is a value, or e' is error, or $e' \xrightarrow{p} e''$ for some e'' .*

As usual, soundness can be derived from progress and subject reduction properties.

Theorem 2 (Progress). *If $\emptyset \vdash p$ and $\Delta^p; \emptyset; \emptyset; \emptyset \vdash e: T$, then either e is a value, or e is error, or $e \xrightarrow{p} e'$ for some e' .*

Theorem 3 (Subject reduction). *If $\emptyset \vdash p$, $\Delta^p; \emptyset; \emptyset; \emptyset \vdash e: T$, and $e \xrightarrow{p} e'$, then $\Delta^p; \emptyset; \emptyset; \emptyset \vdash e': T'$ for some T' s.t. $\Delta^p \vdash T' \leq T$.*

The proofs are a simplified version of those for FJ. Indeed, METAFJIG programs can be roughly seen as FJ programs with *no inheritance* and two primitive types.

4. Checked compile-time execution

We consider now *meta-programs*, that is, sequences of class declarations where arbitrary expressions, rather than class constants, are associated to class names. A meta-program can be reduced to a program by a process that we call *compile-time execution*, formally modeled by the relation $mp \longrightarrow mp'$ defined in Figure 7. As modeled by (META-RED), the right-hand-side of a class declaration can be reduced in the context of the program part of the current meta-program.

However, soundness of compile-time execution is not guaranteed, that is, reduction could get stuck, or produce as right-hand-side of a class declaration a value different from a class constant, or a class constant denoting an ill-typed class.

To prevent these error situations, different approaches are possible. In this paper, rather than guaranteeing soundness by (only) static checks, which would require a sophisticated type system with structural types, we propose a simple solution, called *checked compile-time execution*, which integrates meta-reduction with type-checking steps. If typechecking fails, then the program reduces to `errorT`. The advantage is that the technique can be modularly defined²¹ on top of an arbitrary Java-like language.

²⁰ Analogously to implemented interfaces in Java.

²¹ And implemented, as explained in Section 5.

$$\begin{array}{c} \text{(CLIENT-DIRECT)} \frac{c \xrightarrow{*} C'}{C \longrightarrow_p C'} \quad p(C)=c \quad \text{(CLIENT-REFL)} \frac{}{C \longrightarrow_p C} \\ \text{(CLIENT-TRANS)} \frac{C \longrightarrow_p C' \quad C' \longrightarrow_p C''}{C' \longrightarrow_p C''} \\ \text{(DEPEND-DIRECT)} \frac{e \xrightarrow{0} C' \quad C' \longrightarrow_p C'' \quad mp(C)=e \quad C'' \in \text{dom}(mp)}{C \Longrightarrow_{p;mp} C''} \\ \text{(DEPEND-TRANS)} \frac{C \Longrightarrow_{p;mp} C' \quad C' \Longrightarrow_{p;mp} C''}{C \Longrightarrow_{p;mp} C''} \end{array}$$

Figure 8. Clientship and dependency relations

More in detail, during checked compile-time execution each class declaration `class C=e` in the meta-program passes through-out the following states:

1. initial state, no check has been performed yet;
2. we have checked that e is a well-typed expression of type `class`; hence, e can be safely reduced, until getting a class constant c ;
3. we have checked that c denotes a well-typed class.

Checked compile-time execution is formally defined on *configurations* σ , which are either `error` or `errorT` or triples of the form $p; mp; mp'$ where p , mp and mp' are the class declarations in state (3), (2), and (1), respectively, hence the initial configuration for meta-program mp is $\emptyset; \emptyset; mp$. We assume that configurations are *closed*, that is, only class names which are declared can appear in expressions.

In the rules we use the *clientship* \longrightarrow_p and *dependency* $\Longrightarrow_{p;mp}$ relations defined in Figure 8.

Here, $e \xrightarrow{0} C$ means that (a subexpression of) e is of form C or `new C()`. The superscript 0 suggests that occurrences of C in class constants in e are not taken into account. For instance, if e is

```
new C().m({ D n() { return new D(); } })
```

then $e \xrightarrow{0} C$ holds, but not $e \xrightarrow{0} D$.

On the other hand, $c \xrightarrow{*} C$ means that C occurs in (an inner class constant in) c .

Clientship is analogous to the same notion in Java. That is, C is client of C' if C (transitively) uses C' either as type or as instance generator. Note that C' can occur at any inner level in the class constant defining C . Moreover, we assume that any class is client of itself for technical convenience. In order to determine whether a program p is strongly well-typed, all classes p is client of either must be in p itself or already typechecked. In Java, this relation is indeed used when the compiler is invoked to determine the full set of classes to be compiled or present in bytecode form.

Dependency, instead, is a novel notion, defined on class names in a meta-program. Class C may depend on some other class only if its defining expression e is non constant, that is, still needs to be (typechecked and) reduced. In this case, C depends on all the classes which need to be reduced and typechecked in order to determine whether e is well-typed and of type `class`.

Moreover, we use the following abbreviations:

- $\text{closed}(p, e) = \{ C' \mid e \xrightarrow{0} C, C \longrightarrow_p C' \} \subseteq \text{dom}(p)$, hence we can determine whether or not e is a well-typed expression of type `class` w.r.t Δ^p ;

$$\begin{array}{c}
\text{(VAR-T)} \frac{}{_ ; _ ; \Pi ; _ \vdash x : T} \quad \Pi(x) = T \quad \text{(THIS-T)} \frac{}{_ ; _ ; _ ; \overline{C} \vdash \mathbf{this} : C} \quad C \in \overline{C} \\
\\
\text{(INT-FIELD-T)} \frac{}{_ ; \Sigma ; _ ; _ \vdash f : T} \quad \Sigma(f) = T \quad \text{(INT-INVK-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash \overline{e} : \overline{T}}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash m(\overline{e}) : T} \quad \Sigma(m) = \overline{T}' \rightarrow T \\
\Delta \vdash \overline{T} \leq \overline{T}' \\
\\
\text{(CLIENT-FIELD-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e : C}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e.f : T} \quad \text{mtype}(\Delta, C, f) = T \\
\\
\text{(CLIENT-INVK-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e : C}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e.m(\overline{e}) : T} \quad \text{mtype}(\Delta, C, m) = \overline{T}' \rightarrow T \\
\Delta \vdash \overline{T} \leq \overline{T}' \\
\\
\text{(NEW-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash \overline{e} : \overline{T}}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash \mathbf{new } C(\overline{e}) : C} \quad \begin{array}{l} \text{kind}(\Delta, C) = \neg \mathbf{a} \\ \text{ktype}(\Delta, C) = \overline{T}' \\ \Delta \vdash \overline{T} \leq \overline{T}' \end{array} \\
\\
\text{(OBJ-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_i : T_i \quad \forall i \in 1..n}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash C(f_1=e_1 \dots f_n=e_n) : C} \quad \begin{array}{l} \text{mtype}(\Delta, C, f_i) = T'_i \\ \Delta \vdash T_i \leq T'_i \quad \forall i \in 1..n \end{array} \\
\\
\text{(NAME-T)} \frac{}{_ ; _ ; _ ; _ \vdash n : \mathbf{name}} \quad \text{(CLASS-NAME-T)} \frac{}{\Delta ; _ ; _ ; _ \vdash C : \mathbf{class}} \quad C \in \text{names}(\Delta) \\
\\
\text{(SUM-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_1 : \mathbf{class}}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_1[+]e_2 : \mathbf{class}} \quad \text{(RESTRICT-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_1 : \mathbf{class}}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_1[\mathbf{restrict } e_2] : \mathbf{class}} \\
\\
\text{(ALIAS-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_1 : \mathbf{class}}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_1[\mathbf{alias } e_2 \text{ to } e_3] : \mathbf{class}} \quad \text{(REDIRECT-T)} \frac{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_1 : \mathbf{class}}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash e_1[\mathbf{redirect } e_2 \text{ to } e_3] : \mathbf{class}} \\
\\
\text{(ERROR-T)} \frac{}{\Delta ; \Sigma ; \Pi ; \overline{C} \vdash \mathbf{error} : T}
\end{array}$$

Figure 6. Typing rules for expressions

- $\text{closed}(p, p')$ if, for all $C \in \text{dom}(p')$, $\{C' \mid C \rightarrow_{p'} C'\} \subseteq \text{dom}(p)$, hence we can determine whether or not p' is a strongly well-typed program w.r.t Δ^p .

Rules defining checked compile-time execution are given in Figure 9.

Each pair of rules models a normal reduction step and the corresponding abnormal termination.

In rule (META-CHECK), class declaration $\text{class } C=e$ passes from state (1) to (2), since e is of type class w.r.t. the already typechecked program portion p . In rule (META-CHECK-ERROR), a type error is raised in two cases: either there exists $\text{class } C=e$ in state (1) for which we have all the information needed to typecheck e , and e is ill-typed, or we detect a cyclic dependency among classes in state (1), hence there is no hope to be able to typecheck all of them in the future.²²

In rule (META-RED), class declaration $\text{class } C=e$ makes a meta-reduction step w.r.t. the already typechecked program portion p . In rule (META-RED-ERROR), the meta-reduction step raises a composition error.

In rule (CHECK), a program portion p' passes from state (2) to (3), since it is well-typed w.r.t. the already typechecked program portion p . In rule (CHECK-ERROR), p' raises a typechecking error,

$$\begin{array}{c}
\text{(META-CHECK)} \frac{}{p \mid mp \mid \mathbf{class } C=e \ mp' \rightarrow p \mid mp \ \mathbf{class } C=e \ \mid mp'} \quad \Delta^p ; \emptyset ; \emptyset ; \emptyset^0 e : \mathbf{class} \\
\\
\text{(META-CHECK-ERROR)} \frac{}{p \mid p' \mid mp \rightarrow \mathbf{errorT}} \quad \begin{array}{l} mp(C) = e \\ \text{closed}(p, e) \text{ and } \Delta^p ; \emptyset ; \emptyset ; \emptyset^0 e : \mathbf{class} \\ \text{or } \Rightarrow_{p', mp} \text{cyclic} \end{array} \\
\\
\text{(META-RED)} \frac{e \xrightarrow{p} e'}{p \mid \mathbf{class } C=e \ mp \mid mp' \rightarrow p \mid \mathbf{class } C=e' \ mp \mid mp'} \\
\\
\text{(META-RED-ERROR)} \frac{e \xrightarrow{p} \mathbf{error}}{p \mid \mathbf{class } C=e \ mp \mid mp' \rightarrow \mathbf{error}} \\
\\
\text{(CHECK)} \frac{}{p \mid p' \ mp \mid mp' \rightarrow p \ p' \ \mid mp \ \mid mp'} \quad \Delta^p \vdash^* p' \\
\\
\text{(CHECK-ERROR)} \frac{}{p \mid p' \ mp \mid mp' \rightarrow \mathbf{errorT}} \quad \text{closed}(p, p') \text{ and } \Delta^p \not\vdash^* p'
\end{array}$$

Figure 9. Checked compile-time execution

²² For example $\text{class } C = \mathbf{new } C().m()$.

since we have all the needed information to typecheck it, and it is ill-typed.

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

First we give an example of successful reduction. The program

```
∅ | ∅ |
class C={class m(){return{int k(){return 1;}};}}
class D={ int m(){return new E().k();}}
class E=new C().m()
```

reduces by two applications of (META-CHECK) to

```
∅|
class C={class m(){return{int k(){return 1;}};}}
class D={int m(){return new E().k();}}|
class E=new C().m()
```

reduces by (CHECK) to

```
class C={class m(){return{int k(){return 1;}};}}|
class D={int m(){return new E().k();}}|
class E=new C().m()
```

reduces by (META-CHECK) to

```
class C={class m(){return{int k(){return 1;}};}}|
class D={int m(){return new E().k();}}
class E=new C().m()|
∅
```

reduces by (META-RED) to

```
class C={class m(){return{int k(){return 1;}};}}|
class D={int m(){return new E().k();}}
class E={int k(){return 1;}}|
∅
```

reduces by (CHECK) to

```
class C={class m(){return{int k(){return 1;}};}}
class D={int m(){return new E().k();}}
class E={int k(){return 1;}} |
∅|∅
```

Note that, after checking class C, it is not possible to check class D, since it depends on class E whose definition is not a class constant yet. Hence, expression `new C().m()` is checked to be of type `class`. At this point, reduction of this expression can take place, and finally the resulting class constant is checked to be well-typed, together with class D.

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

```
∅| ∅ | class C={} class D=new C().k()
```

reduces by (META-CHECK) to

```
∅ | class C={} | class D=new C().k()
```

reduces by (CHECK) to

```
class C={} | ∅ | class D=new C().k()
```

reduces by (META-CHECK-ERROR) to `errorT`.

Class C is checked, and then expression `new C().k()` is checked to be of type `class`. This is not the case, since class C has no method named `k`. Since `new C().k()` is closed w.r.t. class C an `errorT` is raised.

The program

```
∅ | ∅ |
class C={class m(class x){return x [+]
  {int k(){return 1;}};}}
class D=new C().m({int h(){return new D().k();}})
```

reduces by (META-CHECK) to

```
∅ |
class C={class m(class x){return x [+]
  {int k(){return 1;}};}} |
class D=new C().m({int h(){return new D().k();}})
```

reduces by (CHECK) to

```
class C={class m(class x){return x [+]
  {int k(){return 1;}};}} |
∅ |
class D=new C().m({int h(){return new D().k();}})
```

reduces by (META-CHECK) to

```
class C={class m(class x){return x [+]
  {int k(){return 1;}};}} |
class D=new C().m({int h(){return new D().k();}}) |
∅
```

reduces by (META-RED) to

```
class C={class m(class x){return x [+]
  {int k(){return 1;}};}} |
class D={ int h(){return new D().k();}} [+]
  { int k(){return 1;}} |
∅
```

reduces by (META-RED) to

```
class C={ class m(class x){return x [+]
  { int k(){return 1;}};}} |
class D={ int h(){return new D().k();}
  int k(){return 1;}} |
∅
```

reduces by (CHECK) to

```
class C={ class m(class x){return x [+]
  { int k(){return 1;}};}}
class D={ int h(){return new D().k();}
  int k(){return 1;}} |
∅ |
∅
```

Class C is checked, then the expression `new C().m({ int h(){return new D().k();}})` is checked to be of type `class`, and then reduced. Finally, the resulting class D is checked.

This example also illustrates why only (standard) typechecking is used in (META-CHECK). Indeed, the fact that the expression `new D().k()` is well-typed can only be detected when the result of `{int h(){return new D().k();}}` `+{int k(){return 1;}}` is associated to D.

We can state two significant properties for METAFJIG checked compile-time execution. Theorem 4 states that checked-compile time execution never goes stuck. Theorem 7 states that well-typed meta-code never produces ill-typed code. Whereas the former property is shared with [20, 23], the latter is new, and very important, since it allows the programmer to safely use compiled libraries. This is not granted by other approaches like C++ templates. For example in C++ a template instantiation can raise type errors caused by the code of the template itself.

We say that a configuration σ is a *value* if all class declarations are in state (3), formally $\sigma = p \mid \emptyset \mid \emptyset$. Moreover, $\Delta \vdash^{0} mp : \text{class}$ abbreviates $\Delta \vdash^{0} e : \text{class}$ for all `class _=e` in `mp`, and analogously $\Delta \vdash^{*} mp : \text{class}$.

Theorem 4 (Soundness of checked compile-time execution).

If $\emptyset \mid \emptyset \mid mp \xrightarrow{*} \sigma$, then either σ is a value or $\sigma = \text{error}$ or $\sigma = \text{errorT}$ or $\sigma \rightarrow \sigma'$ for some σ' .

Soundness of checked compile-time execution can be proved, as usual, as a consequence of progress and subject reduction properties, where in the latter the invariant which is preserved by reduction is that, in $p \mid mp \mid mp'$, class declarations in p and mp are in state (3) and (2), respectively, where only well-typedness (not strong well-typedness) is required. Formally, we define $\vdash^0 \sigma$ if either $\sigma = \text{error}$, or $\sigma = \text{errorT}$, or $\sigma = p \mid mp \mid mp'$ and $\emptyset \vdash^0 p$, $\Delta^p \vdash^0 mp:\text{class}$. The judgment trivially holds in an initial configuration $\emptyset \mid \emptyset \mid mp$.

Theorem 5 (Progress of checked compile-time execution).

If $\vdash^0 \sigma$, then either σ is a value or $\sigma = \text{error}$ or $\sigma = \text{errorT}$ or $\sigma \longrightarrow \sigma'$ for some σ' .

Theorem 6 (Subject reduction of checked compile-time execution).

If $\vdash^0 \sigma$ and $\sigma \longrightarrow \sigma'$ then $\vdash^0 \sigma'$.

Theorem 7 (Meta-level soundness). If $\emptyset \vdash^* p$ and $\Delta^p \vdash^* mp:\text{class}$, then

$p \mid mp \mid \emptyset \not\rightarrow^* \text{errorT}$.

Meta-level soundness can also be proved by progress and subject reduction properties, with an invariant analogous to that used above. However, here we require *strong* well-typedness. Formally, we define $\vdash^* \sigma$ if either $\sigma = \text{error}$ or $\sigma = p \mid mp \mid \emptyset$ and $\emptyset \vdash^* p$, $\Delta^p \vdash^* mp:\text{class}$.

Theorem 8 (Meta-level progress). If $\vdash^* \sigma$, then either σ is a value or $\sigma = \text{error}$ or $\sigma \longrightarrow \sigma'$ for some σ' .

In order to prove meta-level subject reduction, we need a different version, referring to strong well-typedness rather than well-typedness, of the subject reduction property stated in Theorem 3. This property is stated in Theorem 10, and its proof is based on Lemma 9, which states that by (successfully) applying a composition operator to strongly well-typed classes we always get a strongly well-typed class.

Lemma 9. If $\emptyset \vdash^* p$, $\Delta^* c_1:CT_1$, and $\Delta^* c_2:CT_2$, then

1. $c_1[+]c_2 \xrightarrow{p} c$ implies $\Delta^* c:CT$,
2. $c_1[\text{restrict } n] \xrightarrow{p} c$ implies $\Delta^* c:CT$,
3. $c_1[\text{redirect } n_1 \text{ to } n_2] \xrightarrow{p} c$ implies $\Delta^* c:CT$,
4. $c_1[\text{alias } n_2 \text{ to } n_1] \xrightarrow{p} c$ implies $\Delta^* c:CT$.

Theorem 10 (Strong subject reduction). If $\emptyset \vdash^* p$, $\Delta^p; \emptyset; \emptyset; \emptyset \vdash^* e:T$, and $e \xrightarrow{p} e'$, then $\Delta^p; \emptyset; \emptyset; \emptyset \vdash^* e':T'$ for some T' s.t. $\Delta^p \vdash T' \leq T$.

Theorem 11 (Meta-level subject reduction). If $\vdash^* \sigma$ and $\sigma \longrightarrow \sigma'$ then $\vdash^* \sigma'$.

5. Implementation

As already mentioned, our approach allows a modular implementation, relying on typechecking and execution of the conventional language. This is effectively shown by our prototype compiler, which is built on top on the standard Java compiler and virtual machine. In this section, we describe in some more detail how this modular implementation works.

We will use the metavariables e_J and p_J for plain Java expressions and programs, respectively; moreover, mp_J stands for the sequence `class $C=e_J$` .

First of all, in order to reuse the standard Java compiler, it is clear that we need a translation step which transforms METAFJIG expressions e and programs p into plain Java expressions e_J and programs p_J , respectively.

Notably, primitive types `class` and `name` are not available in plain Java, but are encoded by two classes `MFJClass` and `MFJName`, respectively. The former offers a method for each composition operator. We denote by \tilde{n} and \tilde{c} the values of type `MFJName` and `MFJClass` which are the Java representation (whose details are not relevant) of a name n and a base class c , respectively. If $p=\text{class } C=\tilde{c}$ is a METAFJIG program, then we denote by \tilde{p} the sequence `class $C=\tilde{c}$` , which is a special case of mp_J .

More precisely, the compilation uses two different translation functions, formally defined in Figure 12 in the Appendix: $\llbracket e \rrbracket_0$ and $\llbracket p \rrbracket_*$. The former translates e into the corresponding e_J , while the latter translates p into the corresponding p_J .

Two different functions are needed since well-typedness of the Java code obtained by the translation should encode METAFJIG well-typedness and METAFJIG strong well-typedness of the source METAFJIG code, respectively. Formally, the two translations ensure that the following properties hold:

- if $\llbracket p \rrbracket_*$ is defined, then $\Delta^p = \Delta^{\llbracket p \rrbracket_*}$,
- $\Delta; \emptyset; \emptyset; \emptyset \vdash^* e:\text{class}$ if and only if $\Delta; \emptyset \vdash \llbracket e \rrbracket_0:\text{MFJClass}$, where $\Delta; \Pi \vdash e_J:T$ is the typing judgment for Java expressions,
- $\Delta \vdash^* p$ if and only if $\Delta \vdash \llbracket p \rrbracket_*$, where $\Delta \vdash p_J$ is the typing judgment for Java programs,
- $\Delta^p \vdash C \leq C'$ if and only if $\Delta^p \vdash \mathcal{I}_C \leq \mathcal{I}_{C'}$, where \mathcal{I}_C is the Java interface name which translates C (see Figure 12 in the Appendix).

Figure 10 contains the implementation oriented version of checked compile-time execution. The rules are analogous to those in Figure 9, except that:

- Configurations are now of the form $p_J \mid mp_J \mid mp$.
- In steps (META-CHECK) and (META-CHECK-ERROR) (first case in the side condition), the METAFJIG expression e is first translated into a Java expression and then passed to the Java compiler. If compilation is successful, then the generated bytecode (source and bytecode form are identified here) is added to the second portion of the configuration.
- In steps (CHECK) and (CHECK-ERROR), the METAFJIG program \tilde{p} is first translated into a Java program and then passed to the Java compiler. If compilation is successful, then the generated bytecode is added to the first portion of the configuration.
- In steps (META-RED) and (META-RED-ERROR), the Java (bytecode) expression e_J is evaluated by the JVM. We use the $\xrightarrow{*}$ arrow to model that JVM execution continues until getting a final result or an exception.

6. Conclusion

We have defined a framework for composing classes, where classes are first-class values and new classes can be derived from existing ones by exploiting the full power of the language itself, used on top of a small set of primitive composition operators, instead of using a fixed mechanism like inheritance. Soundness is guaranteed by a lightweight technique, where class composition errors are detected dynamically, and conventional typing errors are detected by interleaving typechecking with meta-reduction steps. The advantages w.r.t. other meta-programming approaches proposed for Java-like languages can be summarized as follows:

- Programmers do not need to learn esoteric syntax and idioms.
- Errors are detected (either by the normal type-checker or dynamically for composition errors) as soon as they appear, not

$$\begin{array}{c}
\text{(META-CHECK)} \frac{}{p_J \mid mp_J \mid \text{class } C=e \text{ mp} \longrightarrow p_J \mid mp_J \text{ class } C=\llbracket e \rrbracket_0 \mid mp} \quad \Delta^{p_J}; \emptyset \vdash \llbracket e \rrbracket_0:\text{MFJClass} \\
\text{(META-CHECK-ERROR)} \frac{}{p_J \mid \tilde{p} \mid mp_J \longrightarrow \text{errorT}} \quad \begin{array}{l} mp(C) = e \\ \text{closed}(p_J, \llbracket e \rrbracket_0) \text{ and} \\ \Delta^{p_J}; \emptyset \not\vdash \llbracket e \rrbracket_0:\text{MFJClass} \\ \text{or } \Longrightarrow_{p;mp} \text{cyclic} \end{array} \\
\text{(META-RED)} \frac{e_J \xrightarrow{\star} \tilde{c}}{p_J \mid \text{class } C=e_J \text{ mp}_J \mid mp \longrightarrow p_J \mid \text{class } C=\tilde{c} \text{ mp}_J \mid mp} \\
\text{(META-RED-ERROR)} \frac{e_J \xrightarrow{\star} \text{error}}{p_J \mid \text{class } C=e_J \text{ mp}_J \mid mp \longrightarrow \text{error}} \\
\text{(CHECK)} \frac{}{p_J \mid \tilde{p} \text{ mp}_J \mid mp \longrightarrow p_J \llbracket \tilde{p} \rrbracket_{\star} \mid mp_J \mid mp} \quad \Delta^{p_J} \vdash \llbracket \tilde{p} \rrbracket_{\star} \\
\text{(CHECK-ERROR)} \frac{}{p_J \mid \tilde{p} \text{ mp}_J \mid mp \longrightarrow \text{errorT}} \quad \text{closed}(p_J, \tilde{p}) \text{ and } \Delta^{p_J} \not\vdash \llbracket \tilde{p} \rrbracket_{\star}
\end{array}$$

Figure 10. Implementation of checked compile-time execution

only “a posteriori” when the whole source code is generated, hence earlier and allowing more informative error messages.

- Composition errors are modeled by exceptions, hence programmers can customize error messages and error handling (e.g., taking an alternative action when the composition of some classes fails).
- Meta-level soundness (Theorem 7) guarantees that errors found by the normal type-checker are always due to programmers’ code, and not to the code of the template itself. This does not hold, e.g., for C++ templates.

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 [30], Prolog [28], OpenJava [31], and JavaMint [32] are meta-circular languages, while C++ [19], D [10], Meta-trait-Java [26] and MorphJ [17] use a specialized meta-language.²³ Almost any dynamically typed language allows 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. Regarding code generation, JavaMint, 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.²⁴ The work presented in this paper lies in the area of meta-circular compile-time execution.

Among the above mentioned approaches, [31] 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

²³ The latest version of D seems to include a limited form of metacircular compilation.

²⁴ Some dynamic languages like Groovy and Lisp allow also compile-time meta-programming.

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.

The comparison with JavaMint is also interesting, since our work is specular in many ways: JavaMint generates code at run-time, which consists of *expressions* only, and whose type is statically known. Instead, we generate code at compile-time, consisting of *class definitions* only, and typechecking is (partially) dynamic.

We plan to investigate extensions of our framework in two directions: type system refinement and nested classes.

In the current model, typechecking steps during checked compile-time execution always require code to be *closed*, that is, type information on all class names appearing in code must be available. Hence, typing errors can be detected only at this time. We plan to define a constraint-based type system, as in [2], where code can be typechecked separately, that is, even in absence of some used class, allowing earlier error detection.

A feature of Java-like languages whose impact on our approach seems challenging are nested classes. Among non-trivial issues are the generalization of primitive operators and the order of type-checking.

References

- [1] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In Boris Magnusson, editor, *ECOOP’02 - Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 334–367. Springer, 2002.
- [2] 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.
- [3] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of Java with mixins. In E. Bertino, editor, *ECOOP’00 - European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 154–178. Springer, 2000. An extended version is [4].
- [4] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, September 2003. Extended version of [3].
- [5] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Flexible type-safe linking of components for Java-like languages. In *JMLC’06 - Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2006.
- [6] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Comput. Lang. Syst. Struct.*, 34(2-3):83–108, 2008.
- [7] Viviana Bono, Ferruccio Damiani, and Elena Giachino. On traits and types in a Java-like setting. In *TCS’08 - IFIP Int. Conf. on Theoretical Computer Science*. Springer, 2008.
- [8] Gilad Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
- [9] 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.
- [10] Digital Mars. D programming language, 2007. <http://www.digitalmars.com/>.
- [11] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

- [12] Erik Ernst. Family polymorphism. In J.L. Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 303–326. Springer, 2001.
- [13] Erik Ernst. Higher-order hierarchies. In L. Cardelli, editor, *ECOOP'03 - Object-Oriented Programming*, number 2743 in Lecture Notes in Computer Science, pages 303–328. Springer, 2003.
- [14] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Intl. Conf. on Functional Programming 1998*, 1998.
- [15] Kathleen Fisher and John Reppy. A typed calculus of traits. In *FOOL'04 - Intl. Workshop on Foundations of Object Oriented Languages*, 2004.
- [16] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183. ACM Press, 1998.
- [17] 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.
- [18] 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.
- [19] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, 2003.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] Giovanni Lagorio, Marco Servetto, and Elena Zucca. A lightweight approach to customizable composition operators for Java-like classes. *Electronic Notes in Theoretical Computer Science*, 2009. FACS'09 - International Workshop on Formal Aspects of Component Software. To appear.
- [24] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 30(2), 2008.
- [25] Sean McDermid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New age components for old fashioned Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*. ACM Press, 2001. SIGPLAN Notices.
- [26] 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.
- [27] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP'03 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003.
- [28] Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, April 1994.
- [29] Tim Sheard. Accomplishments and research challenges in meta-programming. In *In 2nd Intl. Workshop on Semantics, Applications, and Implementation of Program Generation, LNCS 2196*, pages 2–44. Springer, 2000.
- [30] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

$$\begin{array}{c}
(\leq\text{-SIG}) \frac{\Delta \vdash MT_i \leq MT'_i \forall i \in 1..h}{\Delta \vdash n_1:MT_1 \dots n_k:MT_k \leq n_1:MT'_1 \dots n_h:MT'_h} \quad 1 \leq h \leq k \\
(\leq\text{-METHOD TYPE}) \frac{\Delta \vdash T \leq T' \quad \Delta \vdash \overline{T'} \leq \overline{T}}{\Delta \vdash \overline{T} \rightarrow T \leq \overline{T'} \rightarrow T'} \quad (\leq\text{-REFL}) \frac{}{\Delta \vdash T \leq T} \\
(\leq\text{-DIRECT}) \frac{}{\Delta \vdash C \leq C'} \quad \frac{\Delta(C) = \overline{C}; _ ; _ ; _}{C' \in \overline{C}} \\
(\leq\text{-TRANS}) \frac{\Delta \vdash C \leq C' \quad \Delta \vdash C' \leq C''}{\Delta \vdash C \leq C''}
\end{array}$$

Figure 11. Subtyping

- [31] 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.
- [32] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *PLDI'10 - ACM Conf. on Programming Language Design and Implementation*. ACM Press, 2010.

A. Appendix

Proof 12 (Theorem 5). *If σ is neither a value, nor error, nor errorT , then it has the form $p \mid mp' \mid mp$ where $mp'mp \neq \emptyset$. By cases:*

- Assume first that there exists $\text{class } C = e \in mp'$ with e not a value. Since σ implies $\Delta^p \vdash e : \text{class}$, by Theorem 1 we have that $e \xrightarrow{p} e'$, hence σ reduces by either (META-RED) or (META-RED-ERROR).
- Otherwise, σ has the form $p \mid p' \mid mp$ with $p'mp \neq \emptyset$. Assume $mp = \emptyset$. Then, since σ is closed, $\text{closed}(p, p')$ clearly holds, hence σ reduces by either (CHECK) or (CHECK-ERROR).
- Otherwise, consider the graph with nodes $\text{dom}(mp)$ and an edge from C' into C if $C \xrightarrow{p'; mp} C'$ is a direct dependence, that is, holds by (DEPEND-DIRECT). If the graph is cyclic, then σ reduces by (META-CHECK-ERROR).
- Otherwise, there exists a node C with no entrant edges. That is, there exists $\text{class } C = e \in mp$ s.t. the set $C \equiv \{C'' \mid e \xrightarrow{0} C', C' \xrightarrow{p'} C''\}$ is a subset of $\text{dom}(p, p')$. Assume $C \subseteq \text{dom}(p)$. Then, $\text{closed}(p, e)$ holds, hence σ reduces by either (META-CHECK) or (META-CHECK-ERROR).
- Otherwise, let p'' be the maximal subset of p' s.t. $\text{dom}(p'') \subseteq C$. Clearly $\text{closed}(p, p'')$ holds, hence σ reduces by either (CHECK) or (CHECK-ERROR).

Lemma 13. *If $\emptyset \vdash p$ and $\Delta^p \vdash p'$, then $\emptyset \vdash p p'$.*

Proof 14. *Easy check.*

Proof 15 (Theorem 6).

By cases on the applied rule:

- If the applied rule is (META-CHECK), then the thesis follows from the side condition of the rule.
- If the applied rule is (CHECK), then the thesis follows from the side condition of the rule by Lemma 13.
- If the applied rule is (META-CHECK-ERROR), (META-RED-ERROR) or (CHECK-ERROR), then the thesis trivially holds.

<pre> [[new C(\bar{e})]₀ = [[c]]₀ = [[n]]₀ = [[e₁ + e₂]]₀ = [[e₁ redirect e₂ to e₃]]₀ = [[e₁ alias e₂ to e₃]]₀ = [[e₁ restrict e₂]]₀ = [[C]]₀ = [[class C₁=\tilde{c}_1 ... class C_n=\tilde{c}_n]]_* = [[C, c]]_* = [[C, c]] = </pre>	<pre> new C_C ([[\bar{e}]]₀) \tilde{c} \tilde{n} [[e₁]]₀.sum([[e₂]]₀) [[e₁]]₀.redirect([[e₂]]₀ , [[e₃]]₀) [[e₁]]₀.alias([[e₂]]₀ , [[e₃]]₀) [[e₁]]₀.restrict([[e₂]]₀) C_C.thisRepr [[C₁, \tilde{c}_1]]_* ... [[C_n, \tilde{c}_n]]_* [[C, c]] [[C₁, c₁]] ... [[C_k, c_k]] with extract_*(c) = c₁ ... c_k and C₁ ... C_k fresh public [abstract] class C_C implements I_C{[[C, k]] [[\overline{ad}]] [[\overline{fd}]] [[\overline{md}]] public static MFJClass thisRepr=\tilde{c};} public interface I_C extends I_C{[[\overline{ad}]]' [[\overline{fd}]]' } with c = implements C{k \overline{ad}, \overline{fd}, \overline{md}} and [abstract] is present if ad \neq \emptyset C_C(I_{T₁} x₁ ... I_{T_n} x_n){ f₁ = [[e₁]]₀ ... f_k = [[e_k]]₀ I_T f; public I_T f(){return f;} public abstract I_T f(); public abstract I_T m(I_{T₁} x₁ ... I_{T_n} x_n); I_T m(I_{T₁} x₁ ... I_{T_n} x_n){return [[e]]₀}; I_T f(); I_T f(); I_T m(I_{T₁} x₁ ... I_{T_n} x_n); I_T m(I_{T₁} x₁ ... I_{T_n} x_n); </pre>
<pre> [[C, constructor(T₁ x₁ ... T_n x_n){f₁=e₁ ... f_k=e_k}] = [[T f]] = [[abstract T f]] = [[abstract T m(T₁ x₁ ... T_n x_n);]] = [[T m(T₁ x₁ ... T_n x_n){return e;}]] = [[abstract T f']] = [[T f']] = [[abstract T m(T₁ x₁ ... T_n x_n);]]' = [[T m(T₁ x₁ ... T_n x_n){return e;}]]' = </pre>	<pre> C_C(I_{T₁} x₁ ... I_{T_n} x_n){ f₁ = [[e₁]]₀ ... f_k = [[e_k]]₀ I_T f; public I_T f(){return f;} public abstract I_T f(); public abstract I_T m(I_{T₁} x₁ ... I_{T_n} x_n); I_T m(I_{T₁} x₁ ... I_{T_n} x_n){return [[e]]₀}; I_T f(); I_T f(); I_T m(I_{T₁} x₁ ... I_{T_n} x_n); I_T m(I_{T₁} x₁ ... I_{T_n} x_n); </pre>

Auxiliary functions (informally defined):

- for each METAFJIG class name C , \mathcal{I}_C and \mathcal{C}_C are a Java interface name and class name, respectively, where the functions \mathcal{I} and \mathcal{C} have disjoint codomains.
- $extract_*(\tilde{c})$ extracts all the inner class constants in \tilde{c} .

Figure 12. Translation functions

- If the applied rule is (META-RED), then, since $\Delta^p \vdash^0 e : class$ holds, the thesis follows from Theorem 3.

Proof 16 (Theorem 8).

Since $\vdash^* \sigma$ clearly implies $\vdash^0 \sigma$, by Theorem 5 we know that either $\sigma = errorT$ or the thesis holds. However, $\sigma = errorT$ cannot hold since $\vdash^* \sigma$.

Lemma 17 (Weakening). If $\Delta; \Sigma; \overline{C} \vdash^* md$ then $\Delta, _ ; \Sigma, _ ; \overline{C}, _ \vdash^* md$.

Lemma 18 (Substitution). If $\Delta; \Sigma; \overline{C} \vdash^* md$ then $\Delta; \Sigma[n'/n]; \overline{C} \vdash^* md[n'/n]$.

Proof 19 (Lemma 9).

1. We have that $implements \overline{C}_1\{k_1 \overline{d}_1\} [+] implements \overline{C}_2\{k_2 \overline{d}_2\} \xrightarrow{p} implements \overline{C}\{k \overline{d}\}$ with $\overline{d}_i = \overline{ad}_i \overline{fd}_i \overline{md}_i$.

Since we have applied (CLASS-T), we know that $\Delta; \Sigma^{\overline{fd} \vdash^* k_1}$ and $\Delta; \Sigma^{\overline{ad}_1}; \overline{C}_1 \vdash^* \overline{md}_1$, and analogously for k_2 and \overline{md}_2 . Moreover, since we have applied (SUM), $\overline{d}_1 \parallel \overline{d}_2$ holds, hence $\Sigma^{\overline{ad}_1} \Sigma^{\overline{ad}_2}$ is well-formed, and, by Lemma 17, $\Delta; \Sigma^{\overline{ad}_1} \Sigma^{\overline{ad}_2}; \overline{C}_1 \overline{C}_2 \vdash^* \overline{md}_1 \overline{md}_2$ holds.

Premises for (CONS-T) on k are the union of the premises for (CONS-T) on k_1 and k_2 . The last side condition of (SUM) of the result is the union of the last side conditions of (SUM) of the subcomponents. This implies that $implements \overline{C}\{k \overline{d}\}$ is well-typed by (CLASS-T)

2. We have that $implements \overline{C}_1\{k_1 \overline{d}_1\} [restrict n] \xrightarrow{p} implements \overline{C}\{k \overline{d}\}$ with $\overline{d}_1 = \overline{ad}_1 \overline{fd}_1 \overline{md}_1$. Since we have applied (CLASS-T), we know that $\Delta; \Sigma^{\overline{fd} \vdash^* k_1}$ and $\Delta; \Sigma^{\overline{ad}_1}; \overline{C}_1 \vdash^* \overline{md}_1$. Moreover by (RESTRICT), \overline{d} is obtained from \overline{d}_1 by removing a

member definition, hence $\Sigma^{\overline{d}} = \Sigma^{\overline{d}_1}$. Then, $implements \overline{C}\{k \overline{d}\}$ is well-typed by (CLASS-T).

3. We have that $implements \overline{C}_1\{k \overline{d}_1\} [redirect n_1 to n_2] \xrightarrow{p} implements \overline{C}\{k \overline{d}\}$ with $\overline{d}_1 = \overline{ad}_1 \overline{fd}_1 \overline{md}_1$. Since we have applied (CLASS-T), we know that $\Delta; \Sigma^{\overline{fd} \vdash^* k_1}$ and $\Delta; \Sigma^{\overline{ad}_1}; \overline{C}_1 \vdash^* \overline{md}_1$. Method bodies preserve their types by Lemma 18, the redirect operator removes the abstract declaration for n_1 only if not needed to satisfy the last side condition of (CLASS-T), hence $implements \overline{C}\{k \overline{d}\}$ is well-typed by (CLASS-T).
4. We have $implements \overline{C}_1\{k_1 \overline{d}_1\} [alias n_1 to n_2] \xrightarrow{p} implements \overline{C}\{k \overline{d}\}$ with $\overline{d}_1 = \overline{ad}_1 \overline{fd}_1 \overline{md}_1$.

Since we have applied (CLASS-T), we know that $\Delta; \Sigma^{\overline{fd} \vdash^* k_1}$ and $\Delta; \Sigma^{\overline{ad}_1}; \overline{C}_1 \vdash^* \overline{md}_1$. Moreover, \overline{d} is obtained from \overline{d}_1 by adding one member definition. By side condition of rule (ALIAS), $\Sigma^{\overline{ad}_1} n_2 : mtype(n_1, \overline{md}_1)$ is well-formed, and by Lemma 17 $\Delta; \Sigma^{\overline{ad}_1} n_2 : mtype(n_1, \overline{md}_1); \overline{C}_1 \vdash^* \overline{md}$. The new method/expression n_2 is also well-typed since its body is the same of n_1 , hence we can conclude that $implements \overline{C}\{k \overline{d}\}$ is well-typed by (CLASS-T).

Proof 20 (Theorem 10).

The proof is analogous to that for Theorem 3, by using Lemma 9.

Lemma 21. 1. If $\emptyset \vdash^* p$ and $\Delta^p \vdash^* p'$, then $\emptyset \vdash^* p p'$.

2. If $\Delta^p \vdash^* p' : class$, then $\Delta^p \vdash^* p'$.

Proof 22.

1. Easy check.

2. Since, for all *class* $C=c \in p'$, $\Delta^p \vdash^* c:\text{class}$ has been deduced by rule (CLASS-CONSTANT-T*).

Proof 23 (Theorem 11).

By cases depending on the applied rule:

- (META-CHECK) and (META-CHECK-ERROR) cannot be applied since $\vdash^* p \mid mp \mid mp'$ implies $mp' = \emptyset$.
- If the applied rule is (META-RED), then, since $\Delta^p \vdash^* e:\text{class}$ holds, the thesis follows from Theorem 10.
- If the applied rule is (META-RED-ERROR), then the thesis trivially holds.
- If the applied rule is (CHECK), then the thesis follows from the side condition of the rule by Lemma 21-(1).
- (CHECK-ERROR) cannot be applied by Lemma 21-(2), since $\vdash^* p \mid p' mp \mid mp'$ implies $\Delta^p \vdash^* p':\text{class}$.