Compile-time Reflection and Metaprogramming for Java

Weiyu Miao Jeremy Siek
University of Colorado at Boulder
{weiyu.miao, jeremy.siek}@colorado.edu

Abstract

Java reflection enables us to write reusable programs that are independent of certain classes. However, when runtime performance is a big concern, we propose to use compile-time reflection for writing metaprograms that generate non-reflective class/type specific code, which has lower runtime cost.

We proposed both compile-time reflection and metaprogramming for Java, and extended our previous work on pattern-based traits. Pattern-based traits integrate pattern-based reflection with flexible composition of traits. They are capable of pattern-matching over class members and generating expressive code. We add and formalize pattern-based reflection at the statement-level, which enables a meta program to generate statements. We add reified generics for pattern-based traits, which enables a pattern to iterate over any class when traits are instantiated. We implemented an ORM tool (called PtjORM) using pattern-based traits. PtjORM uses compile-time reflection and our benchmark tests show that it has competitive runtime performance compared to the mainstream Java ORM tools.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definition and Theory; D.3.3 [*Programming languages*]: Language Constructs and Features

General Terms Languages

Keywords compile-time reflection, metaprogramming, object-oriented programming, traits, object-relational mapping

1. Introduction

Java reflection [9] has the ability to inspect and analyze class members (i.e. metadata of a class) at run time. It enables programmers to write reusable programs that are independent of certain classes. For example, we are given a great number of classes and asked to implement the function hasGetters. The function checks if all the fields in a class have getters (i.e. field get methods) and prints out those fields without getters. This function is useful when class objects need to be serialized (e.g. converting objects into XML data) or object-relational mapped: many ORM tools require fields to have getters for access. It is impractical to manually write a hasGetters for each class, because each time we change the fields of a class, we also likely need to change the hasGetters for that class, therefore such implementation is not adaptable to the change of a class.

With runtime reflection, we can write a hasGetters that is applicable to any class. Following shows the partial implementation of the hasGetters.

```
1  public boolean hasGetters(Class cls) {
2      Field[] fds = cls.getFields();
3      for(int i = 0; i < fds.length; ++i) {
4          Field fd = fds[i];
5          String mn = "get"+capitalize(fd.getName());
6          Class fTy = fd.getType();
7          try {
8              Method _m = cls.getMethod(mn, new Class[0]);
9              if (! _m.getReturnType().equals(fTy)) {
10                 ... /* print the message that field fd has no getter */
11             }
12         } catch (Exception e){
13             ... /* print the message that field fd has no getter */
14         }
15     }
16     ...
17 }
```

The hasGetters iterates over all the fields of a class (line 2-3). For each field, it creates the name of the getter (line 5, function capitalize converts the first character to upper case) and searches in the class for a method that has the same name and with no parameters (line 8). If such method is found and its return type is equal to the field's type (line 9), then the method is a getter method.

Though Java runtime reflection enables programmers to write general programs, it has three drawbacks.

First, runtime reflection is not efficient. Suppose a program uses the hasGetters method to check a class that is statically known. Each time we run the program, the function needs to iterate over all the fields and the methods in the class. We quote the following words from the Java's documentation [1]:

> "*Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.*"

That is why many Java ORM (Object Relational Mapping) tools (such as Hibernate, Ebean, etc) which use runtime reflection, raise the concern about the runtime cost of reflection.

Second, Java has no specific language that is dedicated to reflection. It uses objects to store metadata. For ease of use and for expressiveness, we would like to have a meta language, which enables users to inspect and manipulate metadata more easily. For example, the meta language of Garcia's calculus [10] for type-reflective metaprogramming has abundant type-level operators and uses types as terms.

[1] see http://docs.oracle.com/javase/tutorial/reflect/

PEPM '14, January 20–21, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2543728.2543739

Third, Java reflection has little support for code generation. Forman's book [9] mentioned code generation via reflection, but the object-level code is written in string, which has no type safety guarantees. We would like Java to have an object-level language that the compiler can type-check before code generation.

To overcome the drawbacks of runtime reflection, researchers have proposed compile-time reflection for its lower run-time cost and improved safety. Draheim [3] developed reflective program generators with compile-time loops that can iterate over the members of one class to create the members for another class. Fähndrich [6] introduced *pattern-based reflection*, which provides high-level patterns and templates. For better code reuse, researchers integrated pattern-based reflection with different composition mechanisms. Huang [11, 12] introduced MorphJ, which increased the expressiveness and safety guarantees of pattern-based reflection. In MorphJ, reflective patterns reside in mixin-like structures [8]: a reflective pattern can match the members of a parametrized superclass and generate members for its subclass. Mixins support linear composition, which imposes some restriction on code reuse. Borrowing the concept of the reflective patterns from MorphJ, we proposed pattern-based traits [20]. They are the combination of pattern-based reflection with traits [23], which offer more expressive forms of composition.

In this paper, we present pattern-based reflection at the statement level. Pattern-based reflection is not new, but to our knowledge, its use at the statement level has never been fully discussed and formalized. We find out that statement-level pattern-based reflection supports the writing of reusable metaprograms, thus it is necessary to formalize the language and prove that it always generates well-typed statements.

Following shows the `hasGetters` implemented in statement-level pattern-based reflection.

```
1  trait getterChecker<class X>
2  provides {
3      public boolean hasGetters() {
4          boolean has = false; boolean ret = true;
5          pattern <F, name f> F $f in X {
6              pattern <> public F get#$f() in X {
7                  has = true;
8              }
9              if (! has) {
10                 ret = false;
11                 println("Field "+$f::getName()+" has no getter!");
12             } else { has = false; }
13         }
14         return ret;
15     }
16 }
```

Trait getterChecker is parameterized over the class that is to be reflected over. Inside the body of function hasGetters, the outer pattern matches each field in X (line 5) and passes the field's name and the type to the inner pattern, which searches the field's getter method in X (line 6). The symbol # is the name concatenation operator. In line 11, expression $f::getName() is a meta function call that, at compile time, returns the string of field f's name. The hasGetters in the trait can be specialized for any class and also adaptable to the change of a class.

For example, suppose we are given the following class Person:

```
1  class Person {
2      private String full_name;
3      private int age;
4      public int getAge() { return this.age; }
5  }
```

In class Person, field full_name has no getter method. When trait getterChecker is applied to class Person, the trait generates the following hasGetters function:

```
1  public boolean hasGetters() {
2      boolean has = false; boolean ret = true;
3      if (! has) {
4          ret = false;
5          println("Field "+"full_name"+" has no getter!");
6      } else { has = false; }
7      has = true;
8      if (! has) {
9          ret = false;
10         println("Field "+"age"+" has no getter!");
11     } else { has = false; }
12     return ret;
13 }
```

The generated hasGetters function is specified for class Person and it has no runtime reflection in the body. Please note that in line 7, the value of variable has is changed into **true**. So, the second println is not executed.

Besides statement-level pattern-based reflection, we also introduce reified generics, which enables a pattern to iterate over any specific class when type parameters are instantiated. Previously, the set of classes that a pattern-based trait can reflect over includes the class that uses the trait and the class's superclasses. So, we increased expressiveness of pattern-based reflection.

This paper makes the following contributions:

- We introduce pattern-based reflection at the statement level, which supports statement-level code generation and reuse (Section 2.2).
- We add generics to traits and extend ranges with class types, which enable a pattern to reflect over a parametrized class. (Section 2.2).
- We present reified generics, which enable generated code to access specialized types (Section 2.3) . We introduce meta-level functions, which enable users to inspect metadata at the compile time (Section 2.3).
- We formalize our language, both the object language (Section 3) and the meta language, and present their type systems, which are implementable (Section 4).
- We implemented an ORM tool with compile time reflection and benchmark tested the tool to seek the improvement of runtime performance by using compile-time reflection (Section 5).

We implemented our proposed programming language features based on Polyglot[2]. The source code and the ORM tool is available at http://code.google.com/p/pattern-based-traits/.

2. Language Features

In this section, we introduce the language features. We start with an overview of traits and pattern-based reflection. Even though they were introduced in our previous paper [20], it is worthwhile to review their syntax not only for readers' convenience but also for the modification we have made to their syntax.

2.1 Traits

A trait [5, 23] is a unit for code reuse. It provides a set of method implementations, which may depend on class members (i.e. fields or methods) that are not given by the trait. A trait can import/use methods from other traits, which means a trait can be composed of methods imported/used from other traits. When methods are

[2] http://www.cs.cornell.edu/projects/polyglot/

merged during trait composition, we can manipulate those methods (such as method renaming, method exclusion, method aliasing, etc) to avoid name conflicts.

We use the Java trait syntax proposed by Reppy and Turon [22]. A basic trait without parameters has the following syntax:

trait-declaration ::=
  trait *trait-name*
  requires { *requirements* }
  provides { *members* }

The optional **requires** clause contains the signatures of dependent class members. The **provides** clause contains members which include provided methods and the **use** clauses for importing methods from other traits. In detail, the **use** clause contains a list of expressions of trait names and method manipulations.

2.2 Pattern-Based Reflection

A reflective pattern at the statement level resides in a method body. It performs both reflection and code generation. The header part can iterate over a sequence of class members (i.e. fields, methods, and constructors), and pattern-match against each of them. The body contains template code that is parameterized over names and/or types. A reflective pattern generates different code instantiations from different names and types obtained using pattern-matching.

In the following, we give the syntax for reflective patterns at the statement level.

pattern-declaration ::=
  pattern
    <*parameters*> [*modifier-pattern*] *member-pattern*
  in *range*
  { *statements* }

A pattern declaration may have a name, but it is not necessary at the statement level. Pattern parameters include name parameters and (constrained) type parameters.

A member pattern is in the form of a field signature, a method signature, or a constructor signature. It may be prefixed with an access-level modifier pattern for pattern-matching a group of class members with certain access level(s). An access-level modifier pattern can be **public**, **private**, **none** for package-private, **nonprivate** for the access levels that are not private, etc.

A range represents a sequence of class members that a pattern can iterate over. It has the following syntax:

$$range ::= identifier \mid range|\{identifiers\} \mid range\backslash\{identifiers\}$$

which can refer to a class type, a class type variable (including the reserved type variable: **thisType**), a member-selection operation (i.e. selecting specified members from a range), or a member-exclusion operation (i.e. removing specified members from a range). Our previous paper [20] also proposed the sum of two ranges, which is not allowed here.

The body of a statement-level pattern is a sequence of statements. We do not allow a return statement to appear inside a pattern's body, otherwise a pattern can generate statements containing unreachable code.

As we mentioned in the introduction, statement-level reflective patterns enable us to write general methods that are applicable to different classes and also adaptable to class change. For example, when we design an extensible programming language framework, we have to use the visitor pattern for separating the syntax from its behaviors. Following is the syntax of a tiny language for integers and immutable arrays:

$$E ::= Integer \mid E + E \mid \{\overline{E}\} \mid E[E]$$

In the syntax, expression $\{\overline{E}\}$ represents immutable array definition, which is like a sequence of expressions; and $E[E]$ means array access. The below shows the Java code of the language's abstract syntax tree nodes:

```
1  interface Node { void visit(NodeVisitor v); }
2
3  class IntLit extends Node {
4      private int; public void visit(NodeVisitor v) { }
5  }
6  class Addition extends Node {
7      private Node left; private Node right;
8      public void visit(NodeVisitor v) {
9          this.left = v.visit(this.left); this.right = v.visit(this.right);
10     }
11 }
12 class Array extends Node {
13     private List fds;
14     public void visit(NodeVisitor v) {
15         List tmp = new ArrayList();
16         for(Iterator i = this.fds.iterator(); i.hasNext(); ) {
17             Object fd = i.next();
18             if (fd instanceof Node)
19                 tmp.add(v.visit((Node) fd));
20         }
21         this.fds = tmp;
22     }
23 }
24 class ArrAccess extends Node {
25     private Node arr; private Node idx;
26     public void visit(NodeVisitor v) {
27         this.arr = v.visit(this.arr); this.idx = v.visit(this.idx);
28     }
29 }
```

In the above code, we assume that abstract class NodeVisitor gives the method Node visit(Node n) { n.visit(**this**); **return** n; }. For each syntax tree node, we implement the visit method that visits its subnodes. With this visitor pattern, we can easily write a type-checker, an evaluator, and a syntax tree printer for this language (detailed implementations are omitted).

If the above language is fully extended, it will be tedious to manually write a visit method for each syntax node. Therefore, we write the following metaprogram to generate the visit methods.

```
1  trait VisitGen<>
2  provides {
3      public void visit(NodeVisitor v) {
4          pattern<F extends Node, name f> F $f in thisType {
5              this.$f = (F) v.visit(this.$f);
6          }
7          pattern<F extends Collection, name f> F $f in thisType
8          {
9              List tmp = new ArrayList();
10             for(Iterator i = this.$f.iterator(); i.hasNext(); ) {
11                 Object e = i.next();
12                 if (e instanceof Node)
13                     tmp.add(v.visit((Node) e));
14             }
15             this.$f.clear(); this.$f.addAll(tmp);
16         }
17         // insert a reflective pattern for the fields of array types
18         ...
19     }
20 }
```

In trait VisitGen and inside visit, the first pattern (line 4-6) matches the fields which are syntax tree nodes and generates the statements for them. The second pattern (line 7-16) matches and generates statements for the fields of type Collection. In the body of the second pattern, the code traverses the collection and applies the

visit method to an element if it is a syntax tree node. We restrict the range to the type variable **thisType**, which will be automatically substituted for the name of the class that uses VisitGen. In the patterns, we have the code (in lines 5, 10, and 15) that may access the private fields via the **this** variable, and the **thisType** range guarantees the code is well-typed. For instance, when class Addition uses VisitGen, the **thisType** variable is substituted for Addition; a visit method is generated for Addition; and inside the visit method, it is safe to access Addition's private fields. In section 2.4, we give more detailed discussion about type safety.

2.3 Reified Generics

To support code compatibility, Java implements generics using type erasure [3] so that the specialized types are not available at runtime. For instance, the following expressions are not accepted in Java (we assume X is a well-defined type variable): X.**class**, obj **instanceof** X, **new** X(), **new** X[10], etc.

Using metaprogramming, we generate specialized code for the instantiation of generics, like C++ templates [1], so that specialized types are preserved in generated code.

For example, in the following, we give a generic equals function at the meta level, which can be instantiated into a specialized equals function for any class.

```
1  trait EqualGen<>
2  provides {
3      public boolean equals(Object obj) {
4          if (obj instanceof thisType)
5              return equals_k(this, (thisType) obj);
6          return false;
7      }
8      private boolean equals_k(thisType obj1, thisType obj2) {
9          boolean is_equal = true;
10         pattern <primitive T, name f> T $f in thisType {
11             if (obj1.$f != obj2.$f)
12                 is_equal = false;
13         }
14         pattern <T extends Object, name f> T $f in thisType {
15             // code for comparing two objects
16             ...
17         }
18         return is_equal;
19     }
20 }
```

In line 4, the **instanceof** operator is applied to the **thisType** variable. Inside function equals_k, the first pattern matches and compares the fields of primitive types, while the second pattern matches and compares the fields of reference types. When the trait EqualGen is used by some concrete class C, the variable **thisType** is specialized into type C.

Following is the example for another use of reified generics: generating instance creation functions for the support of the factory method pattern [4].

```
1  trait InstanceGen<class X, class S>
2  provides {
3      public S createInstance() throws InstantiationException {
4          X obj = null;
5          pattern <> public constructor() in X {
6              obj = new X();
7          }
8          if (obj instanceof S) {
9              println("An instance of "+X::getName()+" is created.");
10             return (S) obj;
11         }
12         throw new InstantiationException("...");
13     }
14 }
```

The trait receives two arguments: the type of a class whose instance can be created and the interface that the class implements. In line 5-6, the pattern matches a public nullary (or default) constructor for class X and creates an instance via the **new** operator if the constructor exists. In line 8, we check if the created object is an instance of type variable S. In line 9, we have the meta function call X::getName() that returns the name of X at compile time. Suppose we have class Student, which has the nullary constructor and implements interface Person. When trait InstanceGen is applied to Student and Person, it generates the following result:

```
1  public Person createInstance() throws InstantiationException {
2      Student obj = null;
3      obj = new Student();
4      if (obj instanceof Person) {
5          println("An instance of "+"Student"+" is created.");
6          return (Person) obj;
7      }
8      throw new InstantiationException("...");
9  }
```

Besides the meta function getName, we have other predefined meta functions that enable users to inspect metadata at compile time. For instance, expression X::getSimpleName() returns the simple name of type X; X::equals(Y) checks if X is equal to Y; X::isSubType(Y) checks if X is a subtype of Y; X::superClass() returns the direct superclass of X; X::isPrimitive() checks if X is a primitive type; and so on.

2.4 Member Accessilbility

A pattern-based trait uses the pattern structure to reflect over the members of some class A and uses the composition power of traits to extend some class B. We have to discuss the relation between A and B because it has influence over the accessibility of a member. In this section, we discuss the conditions when the members of a class can be accessed via the **this** variable.

Consider the following metaprogram, which generates a function to backup the fields in a superclass.

```
1  trait FieldBackupGen<this-class S>
2  provides {
3      public void backup() {
4          pattern <T, name f> nonprivate T $f in S {
5              pattern <> T backup#$f in thisType {
6                  this.backup#$f = this.$f;
7              }
8          }
9      }
10 }
```

The type variable S in the metaprogram cannot be instantiated with an arbtrary class. For instance, trait FieldBackupGen used as follows generates ill-typed code.

```
1  class Account { protected int balance; }
2  class AccBackup {
3      private int backupBalance;
4      use FieldBackupGen<Account>;
5  }
```

For the above, the body of the instantiated function backup is

```
this.backupBalance = this.balance
```

but field balance cannot be accessed via the **this** variable inside class AccBackup. The correct implementation is to let class AccBackup

[3] see http://docs.oracle.com/javase/tutorial/java/generics/erasure.html

[4] see http://en.wikipedia.org/wiki/Factory_method_pattern

inherit from Account, so we can access Account's non-private members via the **this** variable.

In the definition of trait FieldBackupGen, we use keyword **this-class** to restrict the set of classes that a trait can be applied to. It means that an accepted class must be a super type of the class that uses the trait. In detail, if the trait is used by class A, then an accepted class must be a super class of A or class A itself. Suppose some type variable X is prefixed with **this-class**; the subtype relationship between **thisType** and type variable X is that **thisType** <: X.

In the above code, trait FieldBackupGen's type parameter S is restricted by the **this-class**. Trait FieldBackupGen is used by class AccBackup and is applied to class Account. However, class Account is not a super type of class AccBackup, thus the trait application is rejected by the type system.

For a unrestricted type variable X, we allow users to only access its public members via an instance of X.

3. Calculus for the Object Language

Start from this section, we formalize our language. First, we present the calculus for our object language, which is the modest extension of FJ (short for Featherweight Java) [15] with mutable variables and three basic kinds of statements: variable declaration, assignment, and return statement.

`x`, `y`		term variables
`m`		method name
`f`		field name
`C`		type (i.e. class name)
class decl.	`L` ::=	`class C extends C { C̄ f̄; K M̄ }`
constructor	`K` ::=	`C(C̄ f̄){ super(f̄); this.f̄=f̄; }`
method	`M` ::=	`C m(C̄ x̄){ s; }`
statements	`s` ::=	`return e` | `C x = e; s` | `x = e; s`
expressions	`e` ::=	`x` | `e.f` | `e.m(ē)` | `new C(ē)` | `(C)e` | $\mathbb{E}[\![\mathtt{s}]\!]$

Figure 1. FJ syntax extended with statements.

Figure 1 shows the abstract syntax of the object language. In the figure, the syntax of statements is the addition to FJ. A statement can be a return statement or a sequence of statements with a return statement at the end. The variable declaration `C x = e` declares local variable `x` of type `C` and initializes `x` with the value of expression `e`. The assignment `x = e` assigns the value of `e` to `x`. In Java, an assignment is also an expression that gives the value of `e`, but in our object language, we treat an assignment only as a statement. The rest of the syntax in the figure is the same as FJ except for methods and the expression $\mathbb{E}[\![\mathtt{s}]\!]$. In the object language, the body of a method is a sequence. The expression $\mathbb{E}[\![\mathtt{s}]\!]$ does not appear in a concrete program. It wraps a statement and splices it into an expression.

Statement typing: $\boxed{\Gamma \vdash \mathtt{s} : \mathtt{C}}$

$$\frac{\Gamma \vdash \mathtt{x} : \mathtt{C}_1 \quad \Gamma \vdash \mathtt{e} : \mathtt{C}_0 \quad \mathtt{C}_0 <: \mathtt{C}_1 \quad \Gamma \vdash \mathtt{s} : \mathtt{C}}{\Gamma \vdash \mathtt{x = e; \ s} : \mathtt{C}}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C}_0 \quad \mathtt{C}_0 <: \mathtt{C} \quad \Gamma, \mathtt{x} : \mathtt{C} \vdash \mathtt{s} : \mathtt{C}_1}{\Gamma \vdash \mathtt{C\ x = e; \ s} : \mathtt{C}_1} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C}}{\Gamma \vdash \mathtt{return\ e} : \mathtt{C}}$$

Figure 2. Typing rules for statements.

Figure 2 shows the rules for typing statements. We define Γ as a typing environment, which is a finite mapping from variables to types. An environment extension has the form $\Gamma, \mathtt{x} : \mathtt{C}$, which means extending Γ with variable typing `x : C` only if `x` does not appear in Γ. The judgments $\mathtt{C}_0 <: \mathtt{C}_1$ means class $\mathtt{C}_0$ is a subtype of class $\mathtt{C}_1$; and $\Gamma \vdash \mathtt{e} : \mathtt{C}$ means expression `e` has type `C` under the typing environment Γ. We use the subtyping rules and the typing rules for expressions from FJ's type system (see the figures Fig.1 and Fig.2 in [15]). For the additional expression $\mathbb{E}[\![\mathtt{s}]\!]$, its type should be equal to the type of statement `s`. See the following typing rule:

$$\frac{\Gamma \vdash \mathtt{s} : \mathtt{C}}{\Gamma \vdash \mathbb{E}[\![\mathtt{s}]\!] : \mathtt{C}}$$

We also use the typing rules from FJ to type-check methods and class declarations. For FJ's method typing rule, we need to override the rule of type-checking a method body: we use the statement typing rule to type-check the method body. Due to the space limitation, we do not present those rules in this paper.

Statement reduction: $\boxed{\mathtt{s} \longrightarrow \mathtt{s'}}$

$$\frac{}{\mathtt{C\ x = v; \ s} \longrightarrow [\mathtt{v}/\mathtt{x}]\mathtt{s}} \qquad \frac{}{\mathtt{x = v; \ s} \longrightarrow [\mathtt{v}/\mathtt{x}]\mathtt{s}}$$

$$\frac{\mathtt{e} \longrightarrow \mathtt{e'}}{\mathtt{C\ x = e; \ s} \longrightarrow \mathtt{C\ x = e'; \ s}}$$

$$\frac{\mathtt{e} \longrightarrow \mathtt{e'}}{\mathtt{x = e; \ s} \longrightarrow \mathtt{x = e'; \ s}} \qquad \frac{\mathtt{e} \longrightarrow \mathtt{e'}}{\mathtt{return\ e} \longrightarrow \mathtt{return\ e'}}$$

Figure 3. Statement reduction rules.

$$\begin{array}{lcl}
[\mathtt{e'}/\mathtt{x}](\mathtt{return\ e}) & = & \mathtt{return\ } [\mathtt{e'}/\mathtt{x}]\mathtt{e} \\
[\mathtt{e'}/\mathtt{x}](\mathtt{C\ y = e; \ S}) & = & (\mathtt{C\ y = } [\mathtt{e'}/\mathtt{x}]\mathtt{e; \ } [\mathtt{e'}/\mathtt{x}]\mathtt{S}) \\
[\mathtt{e'}/\mathtt{x}](\mathtt{x = e; \ S}) & = & (\mathtt{x = } [\mathtt{e'}/\mathtt{x}]\mathtt{e; \ S}) \\
[\mathtt{e'}/\mathtt{x}](\mathtt{y = e; \ S}) & = & (\mathtt{y = } [\mathtt{e'}/\mathtt{x}]\mathtt{e; \ } [\mathtt{e'}/\mathtt{x}]\mathtt{S}) \text{ where } \mathtt{x} \neq \mathtt{y}
\end{array}$$

Figure 4. Substitution in statements.

Figure 3 shows the reduction rules for statements. We define `v` as the value of an expression, which is `new C(v̄)`. Similar to $[\mathtt{e'}/\mathtt{x}]\mathtt{e}$, the operation $[\mathtt{e}/\mathtt{x}]\mathtt{s}$ means the substitution of `x` in statement `s` for `e`. Figure 4 shows its definition. The substitution $[\mathtt{e'}/\mathtt{x}](\mathtt{C\ x = e; \ S})$ does not need to be defined. The substitution implies that `x` is declared twice in the same scope, which is precluded by the type system. In the figure, we use the reduction rules for expressions (the form $\mathtt{e} \longrightarrow \mathtt{e'}$) from FJ (see Figure Fig.3 in [15]). But we need to revise the computation rule for method invocation and provide the reduction rules for $\mathbb{E}[\![\mathtt{s}]\!]$. Those rules are shown in Figure 5.

Expression reduction: $\boxed{\mathtt{e} \longrightarrow \mathtt{e'}}$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{s}}{\mathtt{(new\ C(\bar{e}))}.\mathtt{m}(\overline{\mathtt{e_0}}) \longrightarrow \mathbb{E}[\![[\overline{\mathtt{e_0}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\bar{\mathtt{e}})/\mathtt{this}]\mathtt{s}]\!]}$$

$$\frac{\mathtt{s} \longrightarrow \mathtt{s'}}{\mathbb{E}[\![\mathtt{s}]\!] \longrightarrow \mathbb{E}[\![\mathtt{s'}]\!]} \qquad \frac{}{\mathbb{E}[\![\mathtt{return\ v}]\!] \longrightarrow \mathtt{v}}$$

... (The rest of the reduction rules are the same as those for FJ.)

Figure 5. Selected reduction rules for expressions. (Like FJ's *mbody* function, the *mbody* function in the figure returns the parameter(s) and the body of `C`'s method `m`.)

The calculus of our object language is type safe. It inherits the properties of FJ: type preservation and progress for expressions. Besides, the properties of type preservation and progress also apply to statements. See the following properties for statements.

Lemma 1 (Substitution Preserves Typing). *If* $\Gamma, \mathtt{x} : \mathtt{C}_0 \vdash \mathtt{s} : \mathtt{C}$ *and* $\Gamma \vdash \mathtt{e} : \mathtt{C}_1$ *where* $\mathtt{C}_1 <: \mathtt{C}_0$*, then* $\Gamma, \mathtt{x} : \mathtt{C}_0 \vdash [\mathtt{e}/\mathtt{x}]\mathtt{s} : \mathtt{C'}$ *for some* `C'` *such that* $\mathtt{C'} <: \mathtt{C}$.

Please note that, in Lemma 1, when type-checking $[\mathtt{e}/\mathtt{x}]\mathtt{s}$, we do not remove the typing of `x` from typing context Γ, because the substitution of `x` in `s` does not always substitute all the `x`s in `s` (see the third rule in Figure 4). So, variable `x` might be still in $[\mathtt{e}/\mathtt{x}]\mathtt{s}$.

Theorem 1 (Type Preservation). *If* $\Gamma \vdash \mathtt{s} : \mathtt{C}$ *and* $\mathtt{s} \longrightarrow \mathtt{s'}$*, then* $\Gamma \vdash \mathtt{s'} : \mathtt{C'}$ *for some* `C'` *such that* $\mathtt{C'} <: \mathtt{C}$.

Theorem 2 (Progress). *If* $\Gamma \vdash \mathtt{s} : \mathtt{C}$*, then* `s` *is either a statement value (i.e.* `return v`*) or there is some* `s'` *with* $\mathtt{s} \longrightarrow \mathtt{s'}$.

4. Calculus for the Meta Language

Our calculus for the meta language captures the new language features and presents the kernel part of the language, thus we omit some features that were fully discussed in the previous papers, such as the manipulation of trait members (e.g. name exclusion, member aliasing, etc). We also omit type bounds, modifier patterns, and meta-level functions, which are considered as advanced features.

4.1 Kernel Syntax of the Meta Language

Figure 6 shows the syntax of the core meta language. In the figure, a type τ is a class name associated with the signatures of the class's members. It gives both the nominal and the structural representation of a class type, and it is the value of a range. A range variable `X` can be used as a type variable. When the range variable `X` is substituted for some τ in type `T`, `X` should be substituted for the nominal representation of τ, that is the substitution $[\mathtt{C} \diamond \{\overline{\mathtt{F}}\mathtt{;}\mathtt{Q}\mathtt{;}\overline{\mathtt{H}}\}/\mathtt{X}]\mathtt{T}$ is reduced into $[\mathtt{C}/\mathtt{X}]\mathtt{T}$. But it performs a normal substitution when the range variable `X` is substituted for some τ in range `R`.

In the expressions, we can use the `new` operator to create an instance of a parameterized class, for instance, we can write the expression $\mathtt{new\ X}(\overline{\mathtt{e}})$.

A trait `TR` is parameterized over ranges. We attach the symbol $+$ or the symbol $-$ to each range parameter. The parameter $\mathtt{X}^{+}$ means the members (excluding constructors) of `X` can be accessed via the `this` variable. The parameter $\mathtt{X}^{-}$ means the members of `X` can be only accessed via an instance of `X`.

Ranges include member selection ($\mathtt{R}|\{\overline{\mathtt{l}}\}$) and member exclusion ($\mathtt{R}\backslash\{\overline{\mathtt{l}}\}$). A range is evaluated into a range value τ, which does not appear in a concrete program. For the range of some class `C`, we compute its value by collecting the signatures of the members in `C`, including `C`'s inherited members.

4.2 Type System

In this section, we discuss the type system of the meta language calculus. First, we give some preliminary definitions (see Figure 7) which are used by the type system.

In the figure, member types present the types for different kinds of class members: `T` for fields, $\overline{\mathtt{T}} \to \mathtt{T}$ for methods, and $\overline{\mathtt{T}} \to \cdot$ for constructors. A member name ℓ extends `l` with the reserved name $\mathbb{C}$ as the unified name for all constructors. A member typing context Δ is the mapping from names to member types. The notation $\Delta[\ell \mapsto \phi]$ means to extend Δ with the mapping from ℓ to ϕ. The notation $\Delta_1 \oplus \Delta_2$ means to merge two member typing contexts if the domains of Δ_1 and Δ_2 are disjoint; otherwise it generates a type error. A structural typing context Θ is the mapping from structure names to member typing contexts. The notation $\Theta \uplus [\kappa \mapsto \Delta]$ means to extend Θ with the new mapping from κ to Δ if $\kappa \notin \Theta$, or means $(\Theta\backslash\kappa) \uplus [\kappa \mapsto \Delta \oplus \Theta(\kappa)]$ if $\kappa \in \Theta$. A variable binding context Γ can bind a term variable to a nominal type ($\mathtt{x} : \mathtt{T}$), a range variable $\mathtt{X}^{o}$, the `thisType` variable, or a name variable η.

`x`, `y`	term variables
`X`, `Y`	type/range variables
η	member name variable
`m`	method name
`f`	field name
`C`	class name
$\mathbb{C}$	reserved name constant
$\mathcal{T}$	trait name
`L`	object class declaration (see Figure 1)

(1) types, names, and member signatures:

type	$\tau ::= \mathtt{C} \diamond \{\overline{\mathtt{F}}\mathtt{;}\mathtt{Q}\mathtt{;}\overline{\mathtt{H}}\}$
nonvar. types	$\mathtt{N} ::= \mathtt{C} \mid \mathtt{thisType}$
nominal types	$\mathtt{U}, \mathtt{T} ::= \mathtt{X} \mid \mathtt{N}$
member names	$\mathtt{l} ::= \eta \mid \mathtt{f} \mid \mathtt{m}$
field sig.	$\mathtt{F} ::= \mathtt{T\ l}$
constructor sig.	$\mathtt{Q} ::= \mathbb{C}(\overline{\mathtt{T}})$
method sig.	$\mathtt{H} ::= \mathtt{T\ l}(\overline{\mathtt{T}})$

(2) meta classes:

meta class	$\mathtt{MC} ::= \mathtt{L\ uses}\ \overline{\mathtt{E}}$
meta meth.	$\mathtt{M} ::= \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{s;}\ \}$
single stmts.	$\mathtt{ss} ::= \mathtt{T\ x = e} \mid \mathtt{x = e} \mid \mathtt{ps}$
statements	$\mathtt{s} ::= \mathtt{return\ e} \mid \mathtt{ss;\ s}$
expressions	$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.l} \mid \mathtt{e.l}(\overline{\mathtt{e}}) \mid \mathtt{new\ T}(\overline{\mathtt{e}}) \mid \mathtt{(T)e}$

(3) traits:

trait	$\mathtt{TR} ::= \mathtt{trait}\ \mathcal{T}\mathtt{<}\overline{\mathtt{X}^{o}}\mathtt{>\ req\ \{}\ \overline{\mathtt{F}}\mathtt{;}\ \overline{\mathtt{H}}\ \mathtt{\}\ prov\ D}$
access ctrl.	$o ::= + \mid -$
trait body	$\mathtt{D} ::= \{\ \overline{\mathtt{M}}\mathtt{;\ use}\ \overline{\mathtt{E}}\ \}$
trait app.	$\mathtt{E} ::= \mathcal{T}\mathtt{<}\overline{\mathtt{R}}\mathtt{>}$

(4) pattern statements:

pattern stmt.	$\mathtt{ps} ::= \mathtt{pattern<}\overline{\mathtt{X}}\mathtt{,}\overline{\eta}\mathtt{>\ P\ in\ R\ \{\ ns;\ \}}$
patterns	$\mathtt{P} ::= \mathtt{F} \mid \mathtt{Q} \mid \mathtt{H}$
ranges	$\mathtt{R} ::= \mathtt{X} \mid \mathtt{N} \mid \mathtt{R}|\{\overline{\mathtt{l}}\} \mid \mathtt{R}\backslash\{\overline{\mathtt{l}}\} \mid \tau$
non-return stmts.	$\mathtt{ns} ::= \mathtt{ss} \mid \mathtt{ss;\ ns}$

Figure 6. Syntax of the calculus for the meta language.

member types	$\phi ::= \mathtt{T} \mid \overline{\mathtt{T}} \to \mathtt{T} \mid \overline{\mathtt{T}} \to \cdot$
member names	$\ell ::= \mathtt{l} \mid \mathbb{C}$
structure names	$\kappa ::= \mathtt{T} \mid \mathcal{T}$
member typings	$\Delta ::= \cdot \mid \Delta[\ell \mapsto \phi] \mid \Delta \oplus \Delta$
structural typings	$\Theta ::= \cdot \mid \Theta \uplus [\kappa \mapsto \Delta]$
variable bindings	$\Gamma ::= \emptyset \mid \Gamma, \mathtt{x : T} \mid \Gamma, \mathtt{X}^{o} \mid \Gamma, \mathtt{thisType} \mid \Gamma, \eta$

Figure 7. Preliminary definitions for the type system.

We define the function $\mathbb{N}$, which computes the nominal representation of a range: $\mathbb{N}(\mathtt{X}) = \mathtt{X}$, $\mathbb{N}(\mathtt{N}) = \mathtt{N}$, $\mathbb{N}(\mathtt{R}|\{\overline{\mathtt{I}}\}) = \mathbb{N}(\mathtt{R})$, $\mathbb{N}(\mathtt{R}\backslash\{\overline{\mathtt{I}}\}) = \mathbb{N}(\mathtt{R})$, and $\mathbb{N}(\mathtt{C} \diamond \{\ldots\}) = \mathtt{C}$. We define the function δ, which literally translates a member pattern into a single member typing context: $\cdot[\ell \mapsto \phi]$.

Before type-checking a program, the compiler computes the structural type (this is member typings) for each class and each trait in the program, thus the following two structural typing contexts are available if there are no name conflicts. First is the context Θ^{CL} that maps the names of all the classes into their member typings. The members of a class include declared members, members imported from used traits, and members inherited from the superclasses. Second is the context Θ^{TR} that maps the names of all the traits into their member typings. The members of a trait include declared members, members imported from used traits, and members in trait requirement.

Pattern statement typing: $\boxed{\Gamma;\Theta \vdash \mathtt{ps}\ \mathtt{ok}}$

$$\frac{\mathbb{N}(\mathtt{R}) = \mathtt{C} \qquad \Gamma \vdash \mathtt{R}\ \mathtt{ok} \qquad \Gamma' = \Gamma, \overline{\mathtt{X}^-}, \overline{\eta} \quad \Gamma' \vdash \mathtt{P}\ \mathtt{ok} \quad \Theta' = \Theta \uplus [\mathtt{C} \mapsto \delta(\mathtt{P})] \quad \Gamma'; \Theta' \vdash \mathtt{ns}\ \mathtt{ok}}{\Gamma;\Theta \vdash \mathtt{pattern<}\overline{\mathtt{X}}\mathtt{,}\overline{\eta}\mathtt{>\ P\ in\ R\ \{\ ns;\ \}}\ \mathtt{ok}}$$

$$\frac{\begin{array}{c}\mathbb{N}(\mathtt{R}) = \mathtt{X} \qquad \mathtt{X}^+ \in \Gamma \qquad \Gamma' = \Gamma, \overline{\mathtt{X}^-}, \overline{\eta} \qquad \Gamma' \vdash \mathtt{P}\ \mathtt{ok} \qquad \delta(\mathtt{P}) = \Delta \\ (\mathtt{P} = \mathtt{F}\ \text{or}\ \mathtt{P} = \mathtt{H})\ \text{implies}\ \Theta' = \Theta \uplus [\mathtt{X} \mapsto \Delta] \uplus [\mathtt{thisType} \mapsto \Delta] \\ \mathtt{P} = \mathtt{Q}\ \text{implies}\ \Theta' = \Theta \uplus [\mathtt{X} \mapsto \Delta] \qquad \Gamma'; \Theta' \vdash \mathtt{ns}\ \mathtt{ok} \qquad \Gamma \vdash \mathtt{R}\ \mathtt{ok}\end{array}}{\Gamma;\Theta \vdash \mathtt{pattern<}\overline{\mathtt{X}}\mathtt{,}\overline{\eta}\mathtt{>\ P\ in\ R\ \{\ ns;\ \}}\ \mathtt{ok}}$$

$$\frac{\begin{array}{c}(\mathbb{N}(\mathtt{R}) = \mathtt{X}\ \text{and}\ \mathtt{X}^- \in \Gamma)\ \text{or}\ \mathbb{N}(\mathtt{R}) = \mathtt{thisType} \\ \Gamma' = \Gamma, \overline{\mathtt{X}^-}, \overline{\eta} \qquad \Gamma' \vdash \mathtt{P}\ \mathtt{ok} \qquad \Gamma \vdash \mathtt{R}\ \mathtt{ok} \\ \Theta' = \Theta \uplus [\mathbb{N}(\mathtt{R}) \mapsto \delta(\mathtt{P})] \qquad \Gamma'; \Theta' \vdash \mathtt{ns}\ \mathtt{ok}\end{array}}{\Gamma;\Theta \vdash \mathtt{pattern<}\overline{\mathtt{X}}\mathtt{,}\overline{\eta}\mathtt{>\ P\ in\ R\ \{\ ns;\ \}}\ \mathtt{ok}}$$

Statement typing: $\boxed{\Gamma;\Theta \vdash \mathtt{s} : \mathtt{T}}$ and $\boxed{\Gamma;\Theta \vdash \mathtt{ns}\ \mathtt{ok}}$

$$\frac{\Gamma;\Theta \vdash \mathtt{ps}\ \mathtt{ok} \qquad \Gamma;\Theta \vdash \mathtt{s} : \mathtt{T}}{\Gamma;\Theta \vdash \mathtt{ps;\ s} : \mathtt{T}} \qquad \frac{\Gamma;\Theta \vdash \mathtt{e} : \mathtt{T}_0 \qquad \mathtt{T}_0 <: \mathtt{T} \qquad \Gamma, \mathtt{x} : \mathtt{T};\Theta \vdash \mathtt{s} : \mathtt{U}}{\Gamma;\Theta \vdash \mathtt{T\ x = e;\ s} : \mathtt{U}}$$

$$\frac{\Gamma;\Theta \vdash \mathtt{x} : \mathtt{T} \qquad \Gamma;\Theta \vdash \mathtt{e} : \mathtt{T}_0 \qquad \mathtt{T}_0 <: \mathtt{T} \qquad \Gamma;\Theta \vdash \mathtt{s} : \mathtt{U}}{\Gamma;\Theta \vdash \mathtt{x = e;\ s} : \mathtt{U}}$$

$$\frac{\Gamma;\Theta \vdash \mathtt{e} : \mathtt{T}}{\Gamma;\Theta \vdash \mathtt{return\ e} : \mathtt{T}} \qquad \frac{\text{for some } \mathtt{e} \quad \Gamma;\Theta \vdash \mathtt{(ns;\ return\ e)} : \mathtt{T}}{\Gamma;\Theta \vdash \mathtt{ns}\ \mathtt{ok}}$$

Expression typing: $\boxed{\Gamma;\Theta \vdash \mathtt{e} : \mathtt{T}}$

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma;\Theta \vdash \mathtt{x} : \mathtt{T}} \qquad \frac{\Gamma;\Theta \vdash \mathtt{e} : \mathtt{T} \qquad \Gamma \vdash \mathtt{l}\ \mathtt{ok} \qquad \Delta = \Theta(\mathtt{T})}{\Gamma;\Theta \vdash \mathtt{e.l} : \Delta(\mathtt{l})}$$

$$\frac{\begin{array}{c}\Gamma;\Theta \vdash \mathtt{e} : \mathtt{U} \qquad \Gamma \vdash \mathtt{l}\ \mathtt{ok} \qquad \Delta = \Theta(\mathtt{U}) \\ \Delta(\mathtt{l}) = \overline{\mathtt{T}} \to \mathtt{T} \qquad \Gamma;\Theta \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} <: \overline{\mathtt{T}}\end{array}}{\Gamma;\Theta \vdash \mathtt{e.l(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{T}\ \mathtt{ok} \qquad \Delta = \Theta(\mathtt{T}) \qquad \Delta(\mathbb{C}) = \overline{\mathtt{T}} \to \cdot \qquad \Gamma;\Theta \vdash \overline{\mathtt{e}} : \overline{\mathtt{U}} \qquad \overline{\mathtt{U}} <: \overline{\mathtt{T}}}{\Gamma;\Theta \vdash \mathtt{new\ T(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{T}\ \mathtt{ok} \qquad \Gamma;\Theta \vdash \mathtt{e} : \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}\ \text{or}\ \mathtt{T} <: \mathtt{U}}{\Gamma;\Theta \vdash \mathtt{(T)e} : \mathtt{T}} \qquad \frac{\Gamma \vdash \mathtt{T}\ \mathtt{ok} \qquad \Gamma;\Theta \vdash \mathtt{e} : \mathtt{U} \qquad \mathtt{U} \not<: \mathtt{T}\ \text{and}\ \mathtt{T} \not<: \mathtt{U}}{\Gamma;\Theta \vdash \mathtt{(T)e} : \mathtt{T}}\ \textit{stupid warning}$$

Figure 8. Statement and expression typing.

Due to the space limitation, we omit the rules for type-checking ranges ($\Gamma \vdash \mathtt{R}$ ok), member patterns ($\Gamma \vdash \mathtt{P}$ ok), names ($\Gamma \vdash \mathtt{l}$ ok), and nominal types ($\Gamma \vdash \mathtt{T}$ ok). Those rules are not difficult to write. We also omit the subtyping rules ($\mathtt{T} <: \mathtt{T}$). Because we do not have bounded type parameters in the kernel calculus, thus the subtyping rules are similar to those for FJ [15].

In Figure 8, the pattern statement typing has three typing rules. The first rule is for the case when the range R's nominal part is some class name C. In this case, we extend the member typing context of C with the member typing generated from the pattern P: $\Theta \uplus [\mathtt{C} \mapsto \delta(\mathtt{P})]$. The first rule type-checks the following code:

```
1  C obj;
2  pattern <name f> public int $f in C {
3      println($f+" has value "+ obj.$f);
4  }
```

By type-checking the header of the pattern (line 2), C's member typing context is extended with $\mathtt{f} \mapsto \mathbf{int}$, which means C has field f of type **int** within the scope of the pattern's body. So in line 3, the expression obj.$f is well-typed. The second rule is for the case when R's nominal part is some range variable X and $\mathtt{X}^+ \in \Gamma$, thus we know X is a super type of thisType and the members (excluding constructors) of X should be accessible via the this variable. In this case, if P is a field or a method signature, we extend the member typing contexts of both X and thisType with $\delta(\mathtt{P})$; or if P is a constructor signature, then only the member typing context of X is extended, because a constructor is not inheritable. In the following code:

```
1  trait T<this-class X>
2  provides {
3      public m() {
4          pattern <T, name f> public T $f in X {
5              T fx = this.$f;
6          }
7          pattern <> public constructor() in X {
8              X objx = new X();
9              thisType objths = new thisType();      // error
10         }
11     }
12 }
```

The expression of field access in line 5 is well-typed; the instance creation in line 8 is well-typed, but the instance creation in line 9 is not well-typed. The third rule is for the case when R's nominal part is some range variable X with $\mathtt{X}^- \in \Gamma$, or it is the thisType variable. If it is X, then we extend X's member typing context with $\delta(\mathtt{P})$, otherwise we extend the thisType's member typing context with $\delta(\mathtt{P})$. The third rule type-checks the following code:

```
1  trait T<class X> requires { X obj; }
2  provides {
3      public m() {
4          pattern <T, name f> public T $f in X {
5              T fobj1 = obj.$f;
6              T fobj2 = this.$f;      // error
7          }
8          pattern <T, name f> public T $f in thisType {
9              T fths1 = this.$f;
10             T fths2 = obj.$f;       // error
11         }
12     }
13 }
```

Type variables X and thisType are unrelated (i.e. no subtype relations). In the above code, the first pattern only allows us to access f via an instance of X, so the field access in line 6 is not well-typed. The second pattern only allows us to access f via the this variable, so the field access in line 10 is not well-typed.

In FJ, expression typing uses the additional functions for finding a field/method type. In contrast, because the member typing context of a class changes within different environments, thus the rules for

typing expressions in Figure 8 obtain the type of a member via a member typing context.

Method typing: $\boxed{\Gamma;\Theta \vdash \texttt{M ok in N}}$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{\texttt{T}},\texttt{T ok} \quad \Gamma,\overline{\texttt{x}}:\overline{\texttt{T}},\texttt{this}:\texttt{C};\Theta \vdash \texttt{s}:\texttt{U} \quad \texttt{U} <: \texttt{T} \\ \texttt{class C extends } \texttt{C}_0 \texttt{ \{ ... \} uses ...} \\ (\Theta^{CL}(\texttt{C}_0))(\texttt{m}) = \overline{\texttt{U}} \to \texttt{U} \text{ implies } \texttt{U} = \texttt{T} \text{ and } \overline{\texttt{U}} = \overline{\texttt{T}}\end{array}}{\Gamma;\Theta \vdash \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ s; \}} \text{ ok in } \texttt{C}}$$

$$\frac{\Gamma \vdash \overline{\texttt{T}},\texttt{T ok} \quad \Gamma,\overline{\texttt{x}}:\overline{\texttt{T}},\texttt{this}:\texttt{thisType};\Theta \vdash \texttt{s}:\texttt{U} \quad \texttt{U} <: \texttt{T}}{\Gamma;\Theta \vdash \texttt{T m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\texttt{\{ s; \}} \text{ ok in } \texttt{thisType}}$$

Trait application typing: $\boxed{\Gamma;\Theta \vdash \texttt{E ok in N}}$

$$\frac{\begin{array}{c}\texttt{trait } \mathcal{T}\texttt{<}\overline{\texttt{X}^o}\texttt{>} \texttt{ req \{ } \overline{\texttt{F}}\texttt{; } \overline{\texttt{H}} \texttt{ \} prov \{ ... \}} \\ \Gamma \vdash \overline{\texttt{R}} \text{ ok} \quad \Gamma \vdash \overline{\mathbb{N}(\texttt{R})} \text{ in } \texttt{N} \Rightarrow \overline{o} \quad \Theta(\texttt{N}) = \Delta \\ \text{for each } \texttt{T f} \text{ in } \overline{\texttt{F}}: \quad \Delta(\texttt{f}) = \texttt{U} \text{ and } \texttt{U} = \texttt{T} \\ \text{for each } \texttt{T m}(\overline{\texttt{T}}) \text{ in } \overline{\texttt{H}}: \quad \Delta(\texttt{m}) = \overline{\texttt{U}} \to \texttt{U} \text{ and } \texttt{U} = \texttt{T} \text{ and } \overline{\texttt{U}} = \overline{\texttt{T}}\end{array}}{\Gamma;\Theta \vdash \mathcal{T}\texttt{<}\overline{\texttt{R}}\texttt{>} \text{ ok in } \texttt{N}}$$

Trait typing: $\boxed{\texttt{TR ok}}$

$$\frac{\begin{array}{c}\Gamma = \overline{\texttt{X}^o},\texttt{thisType} \quad \Gamma \vdash \overline{\texttt{F}} \text{ ok} \quad \Gamma \vdash \overline{\texttt{H}} \text{ ok} \\ \Theta = \Theta^{CL} \uplus [\texttt{thisType} \mapsto \Theta^{TR}(\mathcal{T})] \\ \Gamma;\Theta \vdash \overline{\texttt{M}} \text{ ok in } \texttt{thisType} \quad \Gamma;\Theta \vdash \overline{\texttt{E}} \text{ ok in } \texttt{thisType}\end{array}}{\texttt{trait } \mathcal{T}\texttt{<}\overline{\texttt{X}^o}\texttt{> req \{ } \overline{\texttt{F}}\texttt{; } \overline{\texttt{H}} \texttt{ \} prov \{ } \overline{\texttt{M}}\texttt{; use } \overline{\texttt{E}} \texttt{ \} ok}}$$

Meta class typing: $\boxed{\texttt{MC ok}}$

$$\frac{\begin{array}{c}\texttt{K} = \texttt{C}(\overline{\texttt{C}_0}\ \overline{\texttt{f}_0},\ \overline{\texttt{C}}\ \overline{\texttt{f}})\texttt{\{super(}\overline{\texttt{f}_0}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;\}} \\ \textit{fields}(\texttt{C}_0) = \overline{\texttt{C}_0}\ \overline{\texttt{f}_0} \quad \emptyset;\Theta^{CL} \vdash \overline{\texttt{M}} \text{ ok in } \texttt{C} \quad \emptyset;\Theta^{CL} \vdash \overline{\texttt{E}} \text{ ok in } \texttt{C}\end{array}}{\texttt{class C extends } \texttt{C}_0 \texttt{ \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}} \texttt{ \} uses } \overline{\texttt{E}} \text{ ok}}$$

Figure 9. Method, trait application, trait, and class typing.

Figure 9 shows the rules for typing methods, trait applications, traits, and meta classes.

There are two rules for typing a method. The first rule type-checks a method that resides in some class C, then the type of the this variable is C and the validity of method overriding needs to be checked. The second rule type-checks a method that resides in a trait, then the type of the this variable is the thisType variable.

For the rule of type-checking a trait application, we have function $\Gamma \vdash \overline{\mathbb{N}(\texttt{R})} \text{ in } \texttt{N} \Rightarrow \overline{o}$ that computes the this access control modifiers for a sequence of ranges. According to the definition of function $\mathbb{N}$, the value of $\mathbb{N}(\texttt{R})$ can be either a type variable or a class name. The definition of function $\Gamma \vdash \overline{\mathbb{N}(\texttt{R})} \text{ in } \texttt{N} \Rightarrow \overline{o}$ is shown as follows:

$$\frac{\texttt{X}^o \in \Gamma}{\Gamma \vdash \texttt{X in N} \Rightarrow o} \qquad \frac{\texttt{N} <: \texttt{N}_0}{\Gamma \vdash \texttt{N}_0 \text{ in } \texttt{N} \Rightarrow +} \qquad \frac{\texttt{N} \not<: \texttt{N}_0}{\Gamma \vdash \texttt{N}_0 \text{ in } \texttt{N} \Rightarrow -}$$

The rule for typing a trait application checks if the modifiers of the parameters are consistent with the modifiers of the trait arguments; and also checks if the trait requirements are satisfied in the use context N.

For the rule of typing a trait declaration, we add the thisType variable into the binding context so that the thisType can appear inside a trait. The initial structural typing context is

$$\Theta^{CL} \uplus [\texttt{thisType} \mapsto \Theta^{TR}(\mathcal{T})]$$

so that we can access a member via the this variable in a trait.

For a meta class, we do not allow an imported method to override a method in the meta class or in the meta class's superclasses. Such restriction is checked when the compiler computes the structural typing context for the meta class before type-checking. So in the rule of typing a meta class, we do not need to check if an imported method correctly overrides a method in a superclass.

4.3 Meta Evaluation

The meta evaluation of a metaprogram includes (1) pattern-matching and code generation, and (2) trait flattening (i.e. methods from traits are inlined into classes). Because many previous papers have fully discussed the process of trait flattening [7, 19, 21, 22], we here focus on the former. Because pattern-matching and code generation is performed at the statement level while trait flattening is performed at the member level, there is no side effect if we discuss the evaluate rules for those separately.

A statement-level pattern is evaluated into a sequence of statements. In Figure 10, we present part of the reduction rules for patterns. The rules are for the pattern-matching of fields. Those rules can be applied to the pattern-matching of methods or constructors with some slight modification. In the figure, the first rule is for the case when the pattern matches a field (it can be implemented by the unification algorithm), then it generates a sequence of instantiated statements. If the local variables in a generated statement have name conflicts with other local variables, we perform α renaming to substitute those local variable for fresh ones. The second rule is for the case when the match fails, then the pattern continues to pattern-match the rest fields. The third rule is for the case when there is no field, then the pattern generates an empty statement sequence.

4.4 Soundness

In this section, we give the properties of our meta language. Our purpose is to show that a metaprogram always generates a piece of well-typed code.

Lemma 2 (Staged Type Preservation of Statements)**.** *For some meta-level non-pattern statement* s *(i.e. statement without patterns), if* $\Gamma;\Theta \vdash \texttt{s} : \texttt{C}$ *where* $\mathit{TyVars}(\Gamma) = \emptyset$*, then when applying object-level type-checking to* s*, we have* $\Gamma' \vdash \texttt{s} : \texttt{C}$ *for some object-level typing environment* Γ'*.*

Definition 1 (Class Name Alias $\tilde{\texttt{N}}$)**.** *An alias of a class name, written* $\tilde{\texttt{N}}$*, is a distinct class name. A class name can have multiple name alias.*
(1) An alias of a class name preserves the nominal subtyping relations of that class.
(2) For any class alias $\tilde{\texttt{N}}$*,* $\tilde{\texttt{N}} \notin \mathrm{dom}(\Theta^{CL})$*.*
(3) For some class name alias $\tilde{\texttt{N}}$ *and some* Θ*, if* $\tilde{\texttt{N}}$ *does not appear in the domain of* Θ*, then* $\Theta(\tilde{\texttt{N}}) = \Theta^{CL}(\texttt{N})$*; otherwise because there exists some* Δ *such that* $\tilde{\texttt{N}} \mapsto \Delta \in \Theta$*,* $\Theta(\tilde{\texttt{N}}) = \Delta$*.*

Lemma 3 (Type Substitution Preserves Typing)**.**
i) Suppose **this** : $\texttt{N}_0 \in \Gamma, \Gamma'$*;* $\tilde{\texttt{N}}$ *is an alias of* N*; and* $\tilde{\texttt{N}} \notin \mathrm{dom}(\Theta)$*. If* $\Gamma, \texttt{X}^o, \Gamma';\Theta \vdash \texttt{e} : \texttt{U}$*,* $\Gamma \vdash \texttt{N}$ ok*, and* $\Gamma \vdash \mathbb{N}(\texttt{N})$ in $\texttt{N}_0 \Rightarrow o$*, then* $\Gamma, [\tilde{\texttt{N}}/\texttt{X}]\Gamma';[\tilde{\texttt{N}}/\texttt{X}]\Theta \vdash [\tilde{\texttt{N}}/\texttt{X}]\texttt{e} : [\tilde{\texttt{N}}/\texttt{X}]\texttt{U}$*.*

ii) Suppose **this** : $\texttt{N}_0 \in \Gamma, \Gamma'$*;* $\tilde{\texttt{N}}$ *is an alias of* N*; and* $\tilde{\texttt{N}} \notin \mathrm{dom}(\Theta)$*. If* $\Gamma, \texttt{X}^o, \Gamma';\Theta \vdash \texttt{ns}$ ok*,* $\Gamma \vdash \texttt{N}$ ok*, and* $\Gamma \vdash \mathbb{N}(\texttt{N})$ in $\texttt{N}_0 \Rightarrow o$*, then* $\Gamma, [\tilde{\texttt{N}}/\texttt{X}]\Gamma';[\tilde{\texttt{N}}/\texttt{X}]\Theta \vdash [\tilde{\texttt{N}}/\texttt{X}]\texttt{ns}$ ok*.*

Lemma 4 (Name Substitution Preserves Typing)**.**
i) If $\Gamma, \eta, \Gamma';\Theta \vdash \texttt{e} : \texttt{U}$*, and* $\Gamma \vdash \texttt{l}$ ok*, then* $\Gamma, \Gamma';[\texttt{l}/\eta]\Theta \vdash [\texttt{l}/\eta]\texttt{e} : \texttt{U}$*.*

ii) If $\Gamma, \eta, \Gamma';\Theta \vdash \texttt{ns}$ ok*, and* $\Gamma \vdash \texttt{l}$ ok*, then* $\Gamma, \Gamma';[\texttt{l}/\eta]\Theta \vdash [\texttt{l}/\eta]\texttt{ns}$ ok*.*

Lemma 5 (Statement Concatenation Preserves Typing)**.** *Suppose function lvar collects all the local variables in a sequence of statements. If* $\Gamma;\Theta \vdash \texttt{ns}_1$ ok*,* $\Gamma;\Theta \vdash \texttt{ns}_2$ ok*, and* $\mathit{lvars}(\texttt{ns}_1) \cap \mathit{lvars}(\texttt{ns}_2) = \emptyset$*, then* $\Gamma;\Theta \vdash \texttt{ns}_1\texttt{;ns}_2$ ok*.*

$$\texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{(\texttt{T}_0\ \texttt{f}_0)\texttt{,}\overline{\texttt{F}}\texttt{;Q;}\overline{\texttt{H}}\}\ \{\ \texttt{ns;}\ \} \longrightarrow [\overline{\texttt{l}}/\overline{\eta}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{ns; (pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{\overline{\texttt{F}}\texttt{;Q;}\overline{\texttt{H}}\}\ \{\ \texttt{ns;}\ \}\texttt{)}$$
where $\exists\overline{\texttt{T}}.\exists\overline{\texttt{l}}$ such that $[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{T} = \texttt{T}_0$ and $[\overline{\texttt{l}}/\overline{\eta}]\texttt{f} = \texttt{f}_0$

$$\texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{(\texttt{T}_0\ \texttt{f}_0)\texttt{,}\overline{\texttt{F}}\texttt{;Q;}\overline{\texttt{H}}\}\ \{\ \texttt{ns;}\ \} \longrightarrow \texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{\overline{\texttt{F}}\texttt{;Q;}\overline{\texttt{H}}\}\ \{\ \texttt{ns;}\ \}$$
where $\nexists\overline{\texttt{T}}.\nexists\overline{\texttt{l}}$ such that $[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{T} = \texttt{T}_0$ and $[\overline{\texttt{l}}/\overline{\eta}]\texttt{f} = \texttt{f}_0$

$$\texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> T f in C} \diamond \{\texttt{[];Q;}\overline{\texttt{H}}\}\ \{\ \texttt{ns;}\ \} \longrightarrow \texttt{[]}$$

Figure 10. Pattern-matching and code generation using a field pattern.

Theorem 3 (Type Preservation for Patterns)**.** *If* $\Gamma;\Theta \vdash \texttt{ps}$ ok *and* $\texttt{ps} \longrightarrow \texttt{ns}$*, then* $\Gamma;\Theta \vdash \texttt{ns}$ ok

Theorem 4 (Progress for Patterns)**.** *If* $\emptyset;\Theta \vdash \texttt{ps}$ ok *and* ps *is in the form of* $\texttt{pattern<}\overline{\texttt{X}}\texttt{,}\overline{\eta}\texttt{> P in R}\ \{\ \texttt{ns}_0\ \}$*, then there is some* ns *with* $\texttt{ps} \longrightarrow \texttt{ns}$ *(suppose* ns *includes an empty list of statements).*

5. ORM via Compile-Time Reflection

Object-relational mapping (ORM for short) is the ability of mapping objects from/to records in a relational database. Some Java ORM tools, such as Hibernate [5], use run-time reflection, which may cause some runtime performance overhead.

We present an ORM tool, called PtjORM, which is a real-world application of pattern-based traits. PtjORM is not as powerful as Hibernate, but it supports compile-time reflection and avoids the overhead of runtime reflection. Like many other ORM tools, PtjORM supports property mapping, association mapping, and inheritance mapping. Current version of PtjORM supports two kinds of inheritance mapping strategies: table per subclass and table per concrete class.

Besides the type of reflection, the performance of an ORM tool can be influenced by other factors, such as object saving/fetching mode and the number of executed SQL statements. So, instead of evaluating compile-time reflection solely, we evaluate the overall performance of PtjORM.

We evaluated PtjORM by testing it with a benchmark based on the 007 benchmark [2]. The 007 benchmark was originally implemented in C++. It tests the performance of persistence for a CAD application. Ibrahim and Cook provided its Java implementation [14]. Our modified 007 benchmark generates the databases of three different sizes: tiny, small, and medium. The tiny one has 996 records; the small one has 11,465 records, and the medium one has 75,779 records. We compare the runtime performance of saving and fetching objects for the following four ORM tools: PtjORM (version 2.0), Hibernate (version 4.1), Ebean (version 2.7.7), and EclipseLink (version 2.4.1). In default, they all use some Java bytecode generation and manipulation (runtime) tool for reflection optimization.

We ran the 007 benchmark on a laptop with a 2.2 GHz, Intel® Core™ i7 processor (4 cores) and 8GB memory. The MySQL database was installed on a desktop with a 1.80GHz Intel® Pentium™ E2160 dual processor and 1GB memory. The desktop and the laptop were connected in the same local network.

5.1 Object Saving Evaluation

First, we give Table 1, which shows the number of the SQL statements executed during the object saving tests. While the others used the table per hierarchy strategy, PtjORM used the table per subclass strategy for inheritance mapping and it generated the largest number of SQL statements.

The ORM tools were tested with MySQL database. The test results about the object-saving performance is show in Table 2. The test results show that even with the largest number of executed SQL statements, PtjORM achieved the best object-saving performance. Approximately, PtjORM is 20% faster than Hibernate, 24% faster than EclipseLink, and 15% faster than Ebean.

ORM tools \ DB size	tiny	small	medium
Ebean	1111	11.7K	75.9K
Eclipselink	1071	11.5K	75.7K
Hibernate	1005	11.6K	75.6K
PtjORM	1125	12.6K	76.7K

Table 1. The number of SQL statements executed by each of the ORM tools for object saving.

ORM tools \ DB size	tiny	small	medium
Ebean	955ms	9610ms	62476ms
Eclipselink	1120ms	10084ms	64357ms
Hibernate	1169ms	9817ms	60178ms
PtjORM	823ms	8258ms	50365ms

Table 2. Time used for object-saving. Unit ms is short for millisecond.

5.2 Object Fetching Evaluation

We use the batch fetching mode for object fetching. For fetching a tiny database, we use batch sizes: 4, 8, 12, 40, 70, and 100. For fetching a small database, we use batch sizes: 40, 100, 160, 400, 700, and 1000. For fetching a medium database, we use batch sizes: 200, 600, 1000, 2000, 3500, and 5000. The tables in Table 3 show the number of SQL statements executed by each of the ORM tools for fetching the databases of three different sizes. From the tables, we note that PtjORM generates the largest number of SQL statements when the batch size is small. With the increase of batch size, the number of SQL statements generated by PtjROM drops dramatically. When fetching a large number of objects, we suggest PtjORM users to assign a large batch size for efficiency. For EclipseLink and Hibernate, increasing the batch size does not have a big influence on the number of SQL statements.

The tables in Table 4 show the time used by each of the ORM tools for fetching the databases of three different sizes. We learned that Ebean has the best performance and one reason is that it generates the least number of SQL statements. For fetching a tiny database, PtjORM is $2\%-70\%$ faster than Hibernate, -2%-70% faster than EclipseLink. When fetching a small database, except for the batch size of 40, PtjORM is $8\%-58\%$ faster than Hibernate; and PtjORM is 60% faster than EclipseLink in average. When fetching a medium database, PtjORM does not perform well with small batch sizes, but with larger batch sizes, its performance exceeds the performance of Hibernate and EclipseLink. With a small batch size, because PtjORM generates a greater number of SQL statements, the cost of executing the SQL statements and commuting with the server outweighs the saving from compile-time reflec-

[5] http://www.hibernate.org

tion. But when the batch size is equal or larger than 3500, its performance exceeds the performance of Hibernate and EclipseLink. In summary, PtjORM has the second best performance for fetching objects. Of course, from the benchmark test results, we feel the need to improve PtjORM by reducing the number of SQL statements that PtjORM generates.

tools \ batch size	4	8	12	40	70	100
Ebean	203	118	91	50	41	37
Eclipselink	159	130	125	149	161	160
Hibernate	228	185	153	142	142	140
PtjORM	458	284	208	79	63	54

tools \ batch size	40	100	160	400	700	1000
Ebean	171	83	65	48	44	41
Eclipselink	431	314	303	290	288	286
Hibernate	428	395	368	335	358	374
PtjORM	976	469	287	141	109	74

tools \ batch size	200	600	1000	2000	3500	5000
Ebean	160	75	58	47	43	41
Eclipselink	454	404	395	389	394	393
Hibernate	615	485	518	527	526	523
PtjORM	1673	648	404	295	178	126

Table 3. The number of SQL statements executed by each of the ORM tools for fetching the databases of three sizes

6. Related Work

For Composition FeatherTrait Java (FTJ) [19] extends Featherweight Java with traits. The FTJ does not support the compile-time reflection but its type system is modular. Chai [25] also extends Java with traits. Chai allows traits to be more generally used: a trait not only performs a building block for a class but also creates a subtype relation with the class. Moreover, when used, a trait can be dynamically substituted by another trait with the same interface. Featherweight Jigsaw [18] applies the concept of traits into classes and therefore, classes are also building blocks with a set of composition operators. Jigsaw uses the symmetric sum for composition. Besides the member-level operations, Jigsaw introduces member modifiers to solve name conflicts. Reppy and Turon [22] introduced the traits that can be parametrized over names, types, and values. These traits offer the same ability of pattern-based reflection and can be used to generate fields and methods for a class. MixML [4] extends ML modules with imports and outports (like require and provide in a trait) for flexible composition. MixML does not flatten a composed module. Instead, it keeps module hierarchy. So, access to a member in a module may need to refer to its namespace.

For Reflection Programming languages such as Genoupe [3], SafeGen [13], CTR [6], and MorphJ [11] support reflection. Genoupe introduces a type system for reflective program generators, which offers an particular high degree of static safety for reflection. However, the type system does not guarantee that generated code is always well-typed. SafeGen uses first-order logic formulae to express patterns and those formulae are used by a theorem prover to check the safety of generated code. Therefore, the type system of SafeGen is undecidable. CTR introduces transforms, which support pattern-based compile-time reflection and static type safety. MorphJ refines the type system of CTR with a modular type system. Considering its power for static reflection, we believe that MorphJ is the calculus closest relative to ours. However, one reflective power of MorphJ that we do not have is negative nested patterns, which can be used to prevent name conflicts. MorphJ cannot extend a class in place while our pattern-based traits can.

For Code Generation Some programming languages, such as MetaML [26], MetaOCaml, and Template Haskell [24], enable metaprogramming by providing multiple stages of computation, where earlier stages can manipulate code for late stages. These languages use explicit stage annotations for the support of code manipulation at expression level. Our programming language supports (two-staged) metaprogramming: pattern-matching, code generation, and trait flattening are performed at compile time while generated program is evaluated at run time. Another difference is that these metapogramming languages do not support compile-time type reflection. In paper [10], Garcia presented a metaprogramming language that has type-reflection at the meta level, but it does not support generics.

Comparison with Aspect-Oriented Programming Aspect-oriented programming (AOP) [17] intends to separate cross-cutting concerns. Though not dedicated for AOP, our pattern-based traits can be used to describe some cross-cutting concerns. Compared with AspectJ [16], which weaves advice code into original application code at the Java bytecode level, pattern-based traits generates source code, which enables a programmer to inspect the effect of a metaprogram. The current stable version of AspectJ, that is AspectJ 5, supports Java generics, but it does not allow the use of type variables and name variables. AspectJ inserts code flexibly, allowing users to define different point cuts for insertion, but pattern-based traits enable us to merely add wrapper code for members.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 12–21, New York, NY, USA, 1993. ACM.

[3] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A type system for reflective program generators. In *Proc. of the Conf. on Generative Programming and Component Engineering*. Springer, 2005.

[4] Derek Dreyer and Andreas Rossberg. Mixin' up the ml module system. In *Proc. of the 13th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2008.

[5] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 2006.

[6] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *Proc. of the Conf. on Generative Programming and Component Engineering*. ACM, 2006.

[7] Kathleen Fisher. A typed calculus of traits. In *In Workshop on Foundations of Object-oriented Programming*, 2004.

[8] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. of the Symposium on Principles of Programming Languages*. ACM, 1998.

[9] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[10] Ronald Garcia and Andrew Lumsdaine. Toward foundations for type-reflective metaprogramming. In *Proc. of the Conf. on Generative Programming and Component Engineering*. ACM, 2009.

[11] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with morphj. In *Proc. of the Conf. on Programming Language Design and Implementation*. ACM, 2008.

tools \ batch size	4	8	12	40	70	100
Ebean	345ms	254ms	217ms	159ms	148ms	140ms
Eclipselink	514ms	484ms	478ms	564ms	632ms	622ms
Hibernate	639ms	599ms	608ms	591ms	625ms	586ms
PtjORM	627ms	458ms	353ms	224ms	195ms	186ms

tools \ batch size	40	100	160	400	700	1000
Ebean	677ms	541ms	509ms	458ms	461ms	456ms
Eclipselink	7964ms	2465ms	2479ms	2390ms	2440ms	2391ms
Hibernate	1505ms	1463ms	1423ms	1365ms	1405ms	1421ms
PtjORM	2167ms	1352ms	1103ms	826ms	697ms	601ms

tools \ batch size	200	600	1000	2000	3500	5000
Ebean	2473ms	2287ms	2299ms	2241ms	2281ms	2285ms
Eclipselink	8337ms	8287ms	8165ms	7704ms	7746ms	7706ms
Hibernate	4970ms	4648ms	4743ms	4952ms	4952ms	5139ms
PtjORM	18365ms	9869ms	6802ms	5310ms	4453ms	4253ms

Table 4. Time used for fetching objects from the databases of three sizes

[12] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer-Verlag, 2007.

[13] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with safegen. *Sci. Comput. Program.*, 2011.

[14] Ali Ibrahim and William R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 50–73, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 2001.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer-Verlag, 2001.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. of the European Conf. on Object-Oriented Programming*. SpringerVerlag, 1997.

[18] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight jigsaw: A minimal core calculus for modular composition of classes. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer-Verlag, 2009.

[19] Luigi Liquori and Arnaud Spiwack. Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 2008.

[20] Weiyu Miao and Jeremy Siek. Pattern-based traits. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1729–1736. ACM, 2012.

[21] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *Journal of Object Technology*, 2006.

[22] John Reppy and Aaron Turon. Metaprogramming with traits. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer, 2007.

[23] Nathanael Schärli, Stphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *Proc. European Conf. on Object-Oriented Programming*. Springer, 2003.

[24] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop '02*. ACM, 2002.

[25] Charles Smith and Sophia Drossopoulou. *Chai: Traits for Java-Like Languages*. In *Proc. of the European Conf. on Object-Oriented Programming*. Springer, 2005.

[26] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proc. of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. ACM, 1997.