An effective translation of *Fickle* into Java*
(extended abstract)

D. Ancona[1], C. Anderson[2], F. Damiani[3],
S. Drossopoulou[2], P. Giannini[4], and E. Zucca[1]

[1] DISI - Università di Genova
[2] Imperial College - London
[3] Dipartimento di Informatica - Università di Torino
[4] DISTA - Università del Piemonte Orientale

Abstract. We present a translation from *Fickle* (a Java-like language allowing dynamic object re-classification, that is, objects that can change their class at run-time) into plain Java. The translation is proved to preserve static and dynamic semantics; moreover, it is shown to be *effective*, in the sense that the translation of a *Fickle* class does not depend on the implementation of used classes, hence can be done in a *separate* way, that is, without having their sources, exactly as it happens for Java compilation. The aim is to demonstrate that an extension of Java supporting dynamic object re-classification could be fully compatible with the existing Java environment.

1 Introduction

Dynamic object re-classification is a feature which allows an object to change its class membership at run-time while retaining its identity. Thus, the object's behavior can change in fundamental ways (*e.g.*, non-empty lists becoming empty, iconified windows being expanded, *etc.*) through re-classification, rather than replacing objects of the old class by objects of the new class. Lack of re-classification primitives has long been recognized as a practical limitation of object-oriented programming. *Fickle* [4] is a Java-like language supporting dynamic object re-classification, aimed at illustrating features for object re-classification which could extend an imperative, typed, class-based, object-oriented language.

Other approaches have been attempted [3, 6, 7]; however, *Fickle* is more within the main stream of the object oriented approach, and moreover it is type-safe, in the sense that any type correct program (in terms of the *Fickle* type system) is guaranteed never to access non-existing fields or methods.

A further problem is how to construct, starting from the *Fickle* design, a working extension with dynamic object re-classification of a real object-oriented language. Java is the first natural candidate to be considered, since *Fickle* can be

* Partially supported by Murst Cofin'99 - TOSCA Project, CNR-GNSAGA, and the EPSRC (Grant Ref: GR/L 76709).

considered a small subset of Java (with only non-abstract classes, instance fields and methods, integer and boolean types and a minimal set of statements and expressions) enriched with features for dynamic object re-classification. Thus, in particular, a $\mathcal{F}$*ickle* class which does not use these features is a Java class.

In this paper, we provide a first important step towards the solution, that is, we show that a Java environment could be easily and naturally extended in such a way to handle standard Java and $\mathcal{F}$*ickle* classes together.

In order to show that, we define a translation from $\mathcal{F}$*ickle* into plain Java. The translation is proved to preserve static and dynamic semantics (that is, well-formed $\mathcal{F}$*ickle* programs are translated into well-formed Java programs which behave "in the same way"). Moreover, the translation is *effective*, in the sense that it gives the basis for an effective extension of a Java compiler. This is ensured by the fact that the translation of a $\mathcal{F}$*ickle* class does not depend on the implementation of used classes, hence can be done in a *separate* way, that is, without having their sources, exactly as it happens for Java compilation. This is so because type information needed by the translation can be retrieved from type information stored in binary files.

Hence, an extension of Java supporting dynamic object re-classification could be fully compatible with the existing Java environment.

The problems we had to solve in order to define a translation that were both manageable from the theoretical and implementative point of view were not trivial. The main issues we had to face were the following:

1. to find an appropriate encoding for re-classifiable objects;
2. to deal with the fact that a standard Java class c can be extended by a re-classifiable class, possibly after c is translated (*i.e.*, compiled);
3. to make the translation as simple as possible, neglecting efficiency in favor of clearer proofs of correctness;
4. to make the translation effective, in the sense that it truly supports separate compilation as in Java.

Concerning point 1), the basic idea of the translation is to represent each re-classifiable $\mathcal{F}$*ickle* object o through a pair $<w, i>$ of Java objects. Roughly speaking, w is a *wrapper* object providing the (non-mutable) *identity* of o, whereas i is an *implementor* object providing the (mutable) *behavior* of o. A re-classification of o changes i but not w, and method invocations are resolved by i.

To solve problems 2), 3) and 4), even non-re-classifiable objects are represented through such a pair $<o, o>$, where o plays both roles. This greatly simplifies the translation, and allows the same treatment for re-classifiable classes (*i.e.*, *state classes* in $\mathcal{F}$*ickle* terminology), and non-re-classifiable classes.

The work presented in this paper comes out of a collaboration among different research groups and is based on their previous experience in the design and implementation of Java extensions [1, 4].

The paper is organized as follows: In Section 2 we introduce $\mathcal{F}$*ickle* informally using an example. In Section 3 we give an informal overview of the translation, while in Section 4 we give the formal description. In Section 5 we state the formal properties of the translation (preservation of static and dynamic semantics) and illustrate the compatibility of the translation with Java separate compilation. In the Conclusion we summarize the relevance of this work and describe further research directions.

A prototype implementation largely based on the translation described in this paper has already been developed [2].

2 *Fickle*: a brief presentation

In this section we introduce *Fickle* informally using an example. However, this section is not intended to be a complete presentation of *Fickle*. We refer to [4] for a complete definition of the language.

For readability, in the examples we allow a slightly more liberal syntax than that used in the formal description of the translation (given in Section 4).

The (extended) *Fickle* program in Fig. 1 defines a class `Stack`, with subclasses `EmptyStack` and `NonEmptyStack`. A stack has a capacity (field `int capacity`) that is, the maximum number of integers it can contain, and the usual operators `isEmpty`, `top`, `push`, and `pop`.

In *Fickle* class definitions may be preceded by the keyword `state` or `root` with the following meaning:

- *state classes* are meant to describe the properties of an object while it satisfies some conditions; when it does not satisfy these conditions any more, it must be explicitly re-classified to another state class. For example, `NonEmptyStack` describes non-empty stacks; if these become empty, then they are re-classified to `EmptyStack`.

  We require state classes to extend either root classes or state classes.
- *root classes* abstract over state classes. Objects of a state class `C1` may be re-classified to a class `C2` only if `C2` is a subclass of the uniquely defined root superclass of `C1`. For example, `Stack` abstracts over `EmptyStack` and `NonEmptyStack`; objects of class `EmptyStack` may be re-classified to `NonEmptyStack`, and vice versa.

  We require root classes to extend only non-root and non-state classes.

Objects of a non-state, non-root class `C` behave like regular Java objects, that is, are never re-classified. However, since state classes can be subclasses of non-state, non-root classes, objects bound to a variable `x` of type `C` *may* be re-classified. Namely, if `C` had two state subclasses `C1` and `C2` and `x` referred to an object o of class `C1`, then o may be re-classified to `C2`.

Objects of an either state or root class `C` are created in the usual way by the expression `new C()`.

```
class StackException extends Exception{
 StackException (String str) {} {super(str);}}
abstract root class Stack{
 int capacity;  // maximum number of elements
 abstract boolean isEmpty()  {};
 abstract int top() {} throws StackException;
 abstract void push(int i) {Stack} throws StackException;
 abstract void pop() {Stack} throws StackException;}
state class EmptyStack extends Stack{
 EmptyStack(int n){} {capacity=n;}
 boolean isEmpty() {} {return true;}
 int top() {} throws StackException {
  throw new StackException("StackUnderflow");}
 void push(int i) {Stack} {
  this!!NonEmptyStack; a=new int[capacity]; t=0; a[0]=i;}
 void pop() {} throws StackException {
  throw new StackException("StackUnderflow");}}
state class NonEmptyStack extends Stack{
 int[] a; // array of elements
 int t;   // index of top element
 NonEmptyStack(int n, int i) {} {capacity=n; a=new int[n]; t=0; a[0]=i;}
 boolean isEmpty() {} {return false;}
 int top() {} {return a[t];}
 void push(int i) {} throw StackException{ t++;
  if (t==capacity) throw new StackException("StackOverflow");
  else a[t]=i; }
 void pop() {Stack} {if (t==0) this!!EmptyStack; else t--;}}
public class StackTest{
 static void main(String[] args) {Stack} throws StackException{
  Stack s=new EmptyStack(100); s.push(3); s.push(5);
  System.out.println(s.isEmpty());
  Stack s1=new NonEmptyStack(100,3); Stack s2=s1; s1.pop();
  System.out.println(s2.isEmpty());}}
```

Fig. 1. Program `StackTest` - stacks with re-classifications

Re-classification statement, `this!!C`, sets the class of `this` to `C`, where `C` must be a state class with the same root class of the static type of `this`. The re-classification operation preserves the types and the values of the fields defined in the root class, removes the other fields, and adds the fields of `C` that are not defined in the root class, initializing them in the usual way. Re-classifications may be caused by re-classification statements, like `this!!NonEmptyStack` in body of method `push` of class `EmptyStack`, or, indirectly, by method calls, like `s.push(3)` in body of `main`. At the start of method `push` of class `EmptyStack` the receiver is an object of class `EmptyStack`, therefore it has the field `capacity`, while it does not have the fields `a` and `t`. After execution of `this!!NonEmptyStack` the receiver is of class NonEmptyStack, the field capacity retains its value while the fields a and t are now available.

Fields, parameters, and values returned by methods (for simplicity, *Fickle* does not have local variables) must have declared types which are not state classes; we call these types *non-state types*. Thus, fields and parameters may denote objects which do change class, but these changes do *not* affect their type. Instead, the type of this *may be a state class* and *may change*.

Annotations like {} and {Stack} before throws clauses and method bodies are called *effects*. Similarly to what happens for exceptions in throws clauses, effects list the root classes of all objects that may be re-classified by execution of that method. Methods annotated by the empty effect {}, like isEmpty, do not cause any re-classification. Methods annotated by non-empty effects, like pop and push by {Stack}, may re-classify objects of (a subclass of) a class in their effect (in the example, of Stack).

A method annotated with effects can be overridden only by methods annotated with the same or less effects[1].

By relying on effects annotations, the type and effect system of *Fickle* ensures that re-classifications will not cause accesses to fields or methods that are not defined for the object.

Note that effects are explicitly declared by the programmer rather then inferred by the compiler. Even though effects inference could be implemented in practice, more flexibility in method overriding can be achieved by allowing the programmer to annotate methods with more effects than those that would be inferred (similarly to what happens for exceptions).

3 An informal overview of the translation

3.1 Encoding *Fickle* objects

The translation is based on the idea that each object o of a state class c can be encoded in Java by a pair $<w, i>$ of objects; we call w the *wrapper object of* i and i the *implementor object of* w. Roughly speaking, w provides the identity and i the behavior of o, so that any re-classification of o changes i but not w and method invocations are resolved by i.

The class of w is called a *wrapper class* and is obtained by translating the root class of c, whereas the class of i is called an *implementor class* and is obtained by translating c. For any pair $<w, i>$ encoding an object of a state class, the class of i is always a proper subclass of the class of w.

An object o which is not an instance of a state class does not need to be encoded in principle; however, the same kind of encoding proposed above can be adopted also in this case, since o can always be encoded by the pair $<o, o>$, where both the wrapper and the implementor are the object o itself (in other words, if c is not a state class, then it may seen as wrapper class of itself). Even

[1] This means that adding a new effect in a method of a class c does not require any change to the subclasses of c, but may require some changes to its superclasses.

though at first sight this may seem inefficient and unnecessary, it allows for a simpler and more effective translation, as explained in the sequel.

The translation of classes follows the following two rules:

- each *Fickle* class is translated into exactly one Java class (including `Object`);
- the translation preserves the inheritance hierarchy.

Throughout the paper we adopt the following terminology:

- the translation of a non-state, non-root class is called a *non-implementor, non-wrapper class*;
- the translation of a root class is called a *wrapper class*;
- the translation of a state class is called an *implementor class*.

We illustrate the above in terms of the example in Fig.1. After the instruction

```
s=new NonEmptyStack(100,3);
```

where `s` has static type `Stack`, the object stored in `s` is encoded in the translation as sketched in Fig.2.

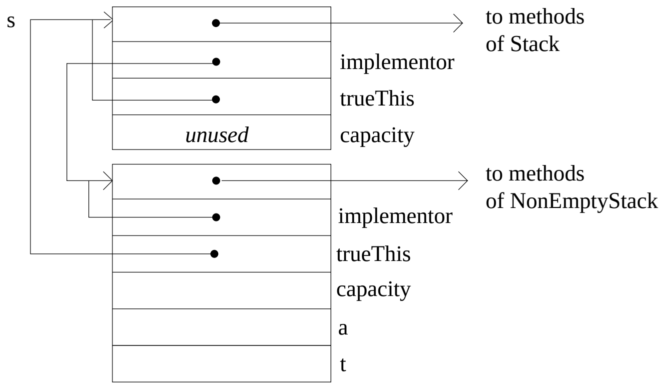

Fig. 2. Encoding of the object stored in `s`

The variable `s` contains an object o of dynamic type `Stack` with three fields: `capacity` is declared in `Stack`, whereas `implementor` and `trueThis` are inherited from class `FickleObject`, have type `FickleObject` and are used in the translation for recovering the implementor and the wrapper of a re-classifiable object, respectively. In this case the field `implementor` points to an object of the implementor class obtained by translating `NonEmptyStack`, whereas `trueThis` points to the object itself. Note that here the field `capacity` is redundant, since its actual value is stored in `implementor.capacity`.

The implementor object contains all fields declared in `NonEmptyStack` (`a` and `t`), and also the field `capacity`, since the implementor class `NonEmptyStack` is a subclass of the wrapper class `Stack`. The field `implementor` points to itself, even though is never used. The field `trueThis` is inherited from class `FickleObject`, has type `FickleObject` and is used to recover the wrapper object of the implementor, which is essential for correctly handling re-classification of `this`.

3.2 Translation of classes

In this section we introduce some examples in order to explain how classes and expressions are translated.

Example 1. Consider the following class declaration in (extended) $\mathcal{F}$*ickle*:

```
class C{
 int x;
 int m1(){}{m2(); return m2();}
 int m2(){R}{x=0; return x;}
}
```

Our translation maps the declaration of `C` in the following Java class[2]

```
class C extends FickleObject{
 int x;
 int m1(){
  ((C) trueThis.implementor).m2();
  return ((C) trueThis.implementor).m2();}
 int m2(){
  ((C) trueThis.implementor).x=0;
  return ((C) trueThis.implementor).x;}
 C(){}
 C(FickleObject oldImp){
  super(oldImp);
  x=((C) oldImp).x;}
}
```

`FickleObject` is the common ancestor of the Java classes obtained by translating $\mathcal{F}$*ickle* classes, and, in fact, corresponds to the translation of the $\mathcal{F}$*ickle* predefined class `Object`:

```
class FickleObject extends Object{
 FickleObject implementor;
 FickleObject trueThis;
 FickleObject(){
  implementor=this;
  trueThis=this;}
```

[2] The translation examples in this paper do not completely agree with the formal definition given in Sect.4, since some optimization has been performed in order to keep the code simpler.

```
 FickleObject(FickleObject oldImp){ // re-classifies objects
  implementor=this;
  trueThis=oldImp.trueThis;
  trueThis.implementor=this;}
}
```

The fields `implementor` and `trueThis` are declared in this top level class for correctly dealing with the encoding of objects which are not instances of state classes, as already explained in 3.1; constructor `FickleObject()` initializes fields `implementor` and `trueThis` to the new instance o so that its encoding is $<o, o>$. This constructor is invoked whenever either a new instance of a non-state class or a new implementor of a state class is created.

On the other hand, constructor `FickleObject(FickleObject oldImp)` is invoked whenever an object is re-classified and is placed in `FickleObject` just for avoiding code duplication. An object o which needs to be re-classified to a state class `C` (recall that in the translation every class is subclass of `FickleObject`) and which is encoded by the pair $<w, i>$, is transformed into $<w, i'>$, where i' denotes the new implementor of class `C` (provided by a proper constructor of `C`; see Example 3 below). The argument of the constructor denotes the old implementor i, from which the wrapper w can be recovered as well (recall that w.`implementor` $= i$.`trueThis` must hold), whereas i' is denoted by `this`. Fields are initialized so that wrapper w and the new implementor i' point to each other. The assignment `implementor=this` could be omitted, since in principle field `implementor` of implementors should never be used.

Two interesting parts of `C` translation concern invocations of method `m2` in `m1` and access of field `x` in `m2`.

Method `m2` must be invoked on `implementor` because it could be overridden by some state subclass of `C`, whereas `this` must be translated in `trueThis` because method `m2` could be inherited by some subclass of `C` (hence, `this` could contain a possibly wrong implementor rather than a wrapper). Downcasting is needed since `implementor` has type `FickleObject`.

The same explanations apply also for selection of field `x`.

Constructor `C(FickleObject oldImp)` invokes the corresponding constructor in class `FickleObject` which is used for re-classifying objects, as already explained. However, during re-classification all fields of the new implementor i' which are inherited from non-state classes (like `x` in the example) must be initialized with the values of the corresponding fields of the old implementor i (`x=((C) oldImp).x`).

Finally, note that the translation of `C` is totally independent of any possible existing subclass or client class of `C`; this property, which is satisfied by our translation for any kind of class, is crucial for obtaining a translation which truly reflects Java separate compilation (see also the related comments in Example 3).

Example 2. Assume now to add to the declaration of Example 1 the following class declaration:

```
root class R extends C{
```

```
}
```

This *Fickle* class declaration is translated in the following Java class declaration:

```
class R extends C{
 R(){}
 R(FickleObject oldImp){super(oldImp);}
 R(R imp){
  trueThis=this;
  implementor=imp;
  imp.trueThis=this;}
}
```

In the translation, root classes declare three constructors.

Constructor `R()` is used for creating instances of `R` and simply invokes the corresponding constructor of the direct superclass `C`.

Constructor `R(FickleObject oldImp)` is used for re-classifying objects and simply invokes the corresponding constructor of the direct superclass `C`, since in this case `R` does not declare any field.

Constructor `R(R imp)` is used by state subclasses of `R` for creating new instances. The argument represents the implementor of the object which has been properly created by the constructor of a state subclass of `R`, while the wrapper object is created by the constructor itself. Fields are initialized so that wrapper and implementor point to each other. The assignment `trueThis=this` could be omitted, since field `trueThis` of wrappers will never be used.

Example 3. Consider now the following state classes:

```
state class S1 extends R{
 int m2(){R}{this!!S2;x=1;return x;}
 static void main(String[] args)
  {System.out.println(new S1().m1());}}
state class S2 extends R{
 int y;
 int m2(){R}{y=1;return x+y;}
}
```

They are translated in Java as follows:

```
class S1 extends R{
 int m2(){
  new S2(trueThis.implementor);
  ((S2) trueThis.implementor).x=1;
  return ((S2) trueThis.implementor).x;}
 static void main(String[] args){
  System.out.println(
   ((S1) new R(new S1()).implementor).m1());}
 S1(){}
```

```
 S1(FickleObject oldImp){super(oldImp);}
}
class S2 extends R{
 int y;
 int m2(){
  ((S2) trueThis.implementor).y=1;
  return ((S2) trueThis.implementor).x+
         ((S2) trueThis.implementor).y;}
 S2(){}
 S2(FickleObject oldImp){super(oldImp);}
}
```

In the translation, state classes declare two constructors.

In class `S2`, for instance, constructor `S2()` is used for creating the implementor component of a new instance of `S2`, while constructor `S2(FickleObject oldImp)` is used for re-classifying objects; note that, differently to what happens for non-state classes, no extra-code is added in the body for any field declared in the class (like `y`).

Let us now focus on the translation of object re-classification `this!!S2` (in the body of method `m2` of class `S1`) and on instance creation of class `S1` (in the body of method `main` of class `S1`).

As already explained, for re-classifying an object to class `S2`, the proper constructor of `S2` must be invoked, passing as parameter the current (and soon obsolete) implementor i, denoted by `trueThis.implementor`; then, the constructor creates a new implementor i' (belonging to `S2`), initializes and updates fields so that the wrapper w and the new implementor i' point to each other (recall that the wrapper can be recovered from the old implementor i) and properly initializes all fields inherited from non-state superclasses (like `x`). This last step is performed by invoking all the corresponding constructors of superclasses up to `FickleObject`.

Creation of an instance of `S1` is achieved by invoking the proper constructor of the root class `R` of `S1`; a new implementor, created by invoking the default constructor of `S1`, is passed as parameter to the constructor.

We now consider issues related to the effectiveness of the translation. As already pointed out in Example 1, the translation of a *Fickle* class `C` does not depend on any possible subclass or client of `C`, as happens for Java separate compilation. On the other hand, the translation of class `S1`, for instance, depends on classes `R` and `S2` inherited and used, respectively, by `S1`; for instance, all type casts in the body of `S1` are determined by type-checking `S1` and this process requires to retrieve type information about classes `R` and `S2` (that is, the signature of methods and the inheritance hierarchy). However, the translation of `S1` is clearly independent of the specific bodies of methods of `R` and `S2`.

As a consequence, dependencies computed by our translation process are exactly the same as those computed by the Java compiler. Furthermore, the translation of classes depends only on the inheritance hierarchy and on method signatures; therefore a class c depending on classes $c_1, \ldots, c_n$ could be success-

$$
\begin{array}{l}
p ::= \mathit{class}^* \\
\mathit{class} ::= [\texttt{root} \mid \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\{\mathit{field}^*\ \mathit{meth}^*\} \\
\mathit{field} ::= t\ f \\
\mathit{meth} ::= t\ m(t'\ x)\phi\{\mathit{sl}\ \texttt{return}\ e;\} \\
t ::= \texttt{boolean} \mid \texttt{int} \mid c \\
\phi ::= \{c^*\} \\
\mathit{sl} ::= s^* \\
s ::= \{\mathit{sl}\} \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \mathit{se}; \mid \texttt{this!!}c; \\
\mathit{se} ::= \mathit{var} = e \mid e_1.m(e_2) \mid \texttt{new}\ c() \\
e ::= \mathit{sval} \mid \mathit{var} \mid \texttt{this} \mid \mathit{se} \\
\mathit{var} ::= \texttt{x} \mid e.f \\
\mathit{sval} ::= \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid n
\end{array}
$$

Fig. 3. Syntax of $\mathcal{F}$*ickle*

fully translated in a context where only the binary files of $c_1, \ldots, c_n$ are available, as happens for Java.

4 Formal description of the translation

In this section we give a formal description of the translation. The syntax of the source language is specified in Fig.3. We refer to [4] for the definition of the static semantics of $\mathcal{F}$*ickle* (the type system of $\mathcal{F}$*ickle* can be easily adapted to the subset of Java serving as target for the translation) and of some auxiliary functions used in the sequel.

4.1 Programs

The translation of a $\mathcal{F}$*ickle* program p consists of the translation of all classes declared in p. The classes are translated w.r.t the program p, needed because the translation of expressions depends on their types (in particular, for method invocation and field selection) and on the names of root classes (in particular, constructor invocation and `this`).

$$[\![p]\!]_{prog} \stackrel{\Delta}{=} [\![\mathit{class}_1]\!]_{class}(p) \ldots [\![\mathit{class}_n]\!]_{class}(p) \qquad \text{where } p = \mathit{class}_1 \ldots \mathit{class}_n.$$

4.2 Classes

As already explained, each $\mathcal{F}$*ickle* class c is translated into a single Java class containing the translation of all field and method declarations of c and a number of constructors, used for creating instances and for re-classifying objects.

The translation of fields and methods is independent of the kind of class.

However, translation of non-state non-root classes, root classes and state classes leads to the declaration of different constructors. That is why for each kind of class we give a different translation clause.

Class `Object`*:* This class is translated in `FickleObject` which is the common superclass of all translated classes, already defined in Sect.3.2.

Non-state, non-root classes: These classes are translated by translating all their methods, and by adding two constructors: $c()$ is used for the creation of new instances of c and $c(\texttt{FickleObject oldImp})$ is used for the creation of new implementors when objects of subclasses are re-classified. In this last case all fields of the old implementor `oldImp` which are declared in class c must be copied into the corresponding new implementor created by the constructor (see Example 1 in Sect.3.2). The additional parameter c for the translation of methods is needed to determine the class of `this` inside the bodies.

$$
\begin{array}{l}
[\![\texttt{class } c \texttt{ extends } c'\{t_1\ f_1; \ldots t_m\ f_m;\ meth_1 \ldots meth_n\}]\!]_{\scriptscriptstyle class}(p) \stackrel{\Delta}{=} \\
\texttt{class } c \texttt{ extends } name(c')\{\ [\![t_1\ f_1;]\!]_{\scriptscriptstyle field}(c) \ldots [\![t_m\ f_m;]\!]_{\scriptscriptstyle field}(c) \\
\qquad\qquad\qquad [\![meth_1]\!]_{\scriptscriptstyle meth}(p,c) \ldots [\![meth_n]\!]_{\scriptscriptstyle meth}(p,c) \\
\qquad\qquad\qquad c()\{\} \\
\qquad\qquad\qquad c(c\ \texttt{oldImp})\{ \\
\qquad\qquad\qquad\quad \texttt{super}(\texttt{oldImp}); \\
\qquad\qquad\qquad\quad f_1 = \texttt{oldImp}.f_1; \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\quad f_m = \texttt{oldImp}.f_m;\} \\
\qquad\qquad\qquad \}
\end{array}
$$

The auxiliary function *name* is defined as follows:

$$
name(c) = \begin{cases} \texttt{FickleObject} & \text{if } c = \texttt{Object} \\ c & \text{otherwise} \end{cases}
$$

Root classes: The translation of this kind of classes produces three constructors: $c()$ creates instances of c, $c(\texttt{FickleObject oldImp})$ deals with object re-classification, and $c(c\ \texttt{imp})$ creates wrappers of instances of state classes:

$$
\begin{array}{l}
[\![\texttt{root class } c \texttt{ extends } c'\{t_1\ f_1; \ldots t_m\ f_m;\ meth_1 \ldots meth_n\}]\!]_{\scriptscriptstyle class}(p) \stackrel{\Delta}{=} \\
\texttt{class } c \texttt{ extends } name(c')\{\ [\![t_1\ f_1;]\!]_{\scriptscriptstyle field}(c) \ldots\ [\![t_m\ f_m;]\!]_{\scriptscriptstyle field}(c) \\
\qquad\qquad\qquad [\![meth_1]\!]_{\scriptscriptstyle meth}(p,c) \ldots [\![meth_n]\!]_{\scriptscriptstyle meth}(p,c) \\
\qquad\qquad\qquad c()\{\} \\
\qquad\qquad\qquad c(c\ \texttt{oldImp})\{ \\
\qquad\qquad\qquad\quad \texttt{super}(\texttt{oldImp}); \\
\qquad\qquad\qquad\quad f_1 = \texttt{oldImp}.f_1; \\
\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\quad f_m = \texttt{oldImp}.f_m;\} \\
\qquad\qquad\qquad c(c\ \texttt{imp})\{ \\
\qquad\qquad\qquad\quad \texttt{trueThis} = \texttt{this}; \\
\qquad\qquad\qquad\quad \texttt{implementor} = \texttt{imp}; \\
\qquad\qquad\qquad\quad \texttt{imp.trueThis} = \texttt{this};\} \\
\qquad\qquad\qquad \}
\end{array}
$$

State classes: The translation of this kind of classes produces two constructors: the former (with no arguments) for creating new implementors for new instances of class c, the latter for dealing with object re-classification to c:

$$\begin{array}{l}
[\![\texttt{state class}\ c\ \texttt{extends}\ c'\{\mathit{field}_1 \ldots \mathit{field}_m\ \ \mathit{meth}_1 \ldots \mathit{meth}_n\}]\!]_{\scriptscriptstyle class}(p) \stackrel{\Delta}{=} \\
\texttt{class}\ c\ \texttt{extends}\ \mathit{name}(c')\{\ [\![\mathit{field}_1]\!]_{\scriptscriptstyle field}(c) \ldots [\![\mathit{field}_m]\!]_{\scriptscriptstyle field}(c) \\
\qquad\qquad\qquad\qquad\qquad\quad [\![\mathit{meth}_1]\!]_{\scriptscriptstyle meth}(p, c) \ldots [\![\mathit{meth}_n]\!]_{\scriptscriptstyle meth}(p, c) \\
\qquad\qquad\qquad\qquad\qquad\quad c()\{\} \\
\qquad\qquad\qquad\qquad\qquad\quad c(\texttt{FickleObject oldImp})\{\texttt{super}(\texttt{oldImp})\} \\
\qquad\qquad\qquad\qquad\qquad\quad \}
\end{array}$$

Note that here $\mathit{name}(c') = c'$, since a state class cannot extend class `Object`.

4.3 Fields

Translation of each field f comes equipped with a static method $\texttt{to}f$ used for translating assigments of value v to field f of object `tT` (see the paragraph on expressions translation below), since the implementor of the object `tT` can be correctly selected only after evaluating v.

$$\begin{array}{l}
[\![t\ f;]\!]_{\scriptscriptstyle field}(c) \stackrel{\Delta}{=} \\
\ t\ f; \\
\ \texttt{static}\ t\ \texttt{to}f(\texttt{FickleObject tT}, t\ v)\{\texttt{return}\ ((c)\ \texttt{tT.implementor}) = v;\}
\end{array}$$

4.4 Methods

Translating methods consists of translating their bodies. Effects are omitted, whereas the signatures remain the same. Since the translation of statements and expressions depends on their types, the program p and the environment γ must be passed as parameters to the corresponding translation functions.

Note that the environment γ' used for translating the returned expression e may be different from γ, since execution of sl could re-classify `this`. Furthermore, translation of each method m comes equipped with a static method $\texttt{call}m$ used for translating invocations of m on receiver `tT` and with argument x (see the paragraph on expressions translation below); indeed, the implementor of `tT` can be correctly selected only after evaluating the argument x.

The judgment $p, \gamma \vdash sl : \texttt{void} \parallel c' \parallel \phi'$ is valid (see [4] for the typing rules) whenever sl has type `void` w.r.t. program p and environment γ; c' denotes the type of `this` after evaluating sl, whereas ϕ conservatively estimates the re-classification effect of the evaluation of sl on objects (this last information is never used by our translation). The environment γ defines the type of the parameters and of `this`.

$$\begin{array}{l}
[\![t\ m(t'\ \texttt{x})\phi\{sl\ \texttt{return}\ e;\}]\!]_{\scriptscriptstyle meth}(p, c) \stackrel{\Delta}{=} \\
\ t\ m(t'\ \texttt{x})\{[\![sl]\!]_{\scriptscriptstyle stmts}(p, \gamma)\ \texttt{return}\ [\![e]\!]_{\scriptscriptstyle expr}(p, \gamma');\} \\
\ \texttt{static}\ t\ \texttt{call}m(\texttt{FickleObject tT}, t'\ \texttt{x})\{ \\
\ \ \texttt{return}\ ((c)\ \texttt{tT.implementor}).m(\texttt{x});\}
\end{array}$$

where $\gamma = t'\ \texttt{x}, c\ \texttt{this}, \quad \gamma' = t'\ \texttt{x}, c'\ \texttt{this}, \quad$ and $\ p, \gamma \vdash sl : \texttt{void} \parallel c' \parallel \phi'$

4.5 Statements

Except for object re-classification, all statements are translated by translating their constituent statements or subexpressions. The notation $\gamma[c\ \texttt{this}]$ denotes the environment obtained by updating γ so that it maps `this` to c.

$$[\![s\ sl]\!]_{stmts}(p,\gamma) \triangleq [\![s]\!]_{stmt}(p,\gamma)\ [\![sl]\!]_{stmts}(p,\gamma')$$
$$\text{where } p,\gamma \vdash s : \texttt{void} \parallel c \parallel \phi \text{ and } \gamma' = \gamma[c\ \texttt{this}]$$
$$[\![\{sl\}]\!]_{stmt}(p,\gamma) \triangleq \{[\![sl]\!]_{stmts}(p,\gamma)\}$$
$$[\![\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2]\!]_{stmt}(p,\gamma) \triangleq$$
$$\texttt{if}\ ([\![e]\!]_{expr}(p,\gamma))\ [\![s_1]\!]_{stmt}(p,\gamma')\ \texttt{else}\ [\![s_2]\!]_{stmt}(p,\gamma')$$
$$\text{where}\quad p,\gamma \vdash e : \texttt{boolean} \parallel c_1 \parallel \phi_1, \quad \gamma' = \gamma[c_1\ \texttt{this}]$$
$$[\![se;]\!]_{stmt}(p,\gamma) \triangleq [\![se]\!]_{expr}(p,\gamma);$$

The translation of re-classification to class c consists of the call to the appropriate constructor of class c. The current implementor (`trueThis.implementor`) is passed as parameter to the constructor in order to correctly initialize the fields of the new implementor.

$$[\![\texttt{this!!}c;]\!]_{stmt}(p,\gamma) \triangleq \texttt{new c(trueThis.implementor)};$$

4.6 Expressions

Types of expressions are preserved under the translation, up to state classes: more precisely, if a *Fickle* expression e has type t and t is not a state class, then its type is preserved; otherwise, the type of the translation of e is the root superclass of t. This is formalized and proven in Sect.5.

Simple cases: Values, variables and variables assignment: The translation is straightforward.

$$[\![sval]\!]_{expr}(p,\gamma) \triangleq sval$$
$$[\![x]\!]_{expr}(p,\gamma) \triangleq x$$
$$[\![\texttt{x} = e]\!]_{expr}(p,\gamma) \triangleq \texttt{x} = [\![e]\!]_{expr}(p,\gamma)$$

Field selection: as already explained in Sect.3.1, in the encoding $<w,i>$ of an object o of class c, the fields of o are stored in the implementor object i (belonging to the class obtained by translating c). Therefore, fields can be accessed only through w.`implementor` on object[3] w. Downcasting is needed because field `implementor` has type `FickleObject`.

$$[\![e.f]\!]_{expr}(p,\gamma) \triangleq ((c)\ [\![e]\!]_{expr}(p,\gamma).\texttt{implementor}).f$$
$$\text{where } p,\gamma \vdash e : c \parallel c' \parallel \phi$$

[3] Note that this is necessary only when c is a state class, while in the other cases selection could be performed directly on the object o itself, since $w = i = o$ holds. However, to keep the mapping simpler, we do not make this distinction.

Field assignment: Field f of the wrapper object w denoted by the translation of e_1 is accessed through the implementor of w; however, e_2 could re-classify w, therefore selection w.`implementor` is correct only after evaluating the translation of e_2. This is achieved by invoking the auxiliary static method `to`f.

$$\begin{array}{l}[\![e_1.f = e_2]\!]_{expr}(p, \gamma) \stackrel{\Delta}{=} c.\mathtt{to}f([\![e_1]\!]_{expr}(p, \gamma), [\![e_2]\!]_{expr}(p, \gamma')) \\ \qquad\qquad \text{where } p, \gamma \vdash e_1 : c \parallel c' \parallel \phi, \text{ and } \gamma' = \gamma[c' \; \mathtt{this}]\end{array}$$

Method invocation: The same considerations as for field assignment apply in this case: method call is performed by calling the auxiliary static method `call`m, so that `implementor` field of the receiver is selected only after evaluating the translation of e_2.

$$\begin{array}{l}[\![e_1.m(e_2)]\!]_{expr}(p, \gamma) \stackrel{\Delta}{=} c.\mathtt{call}m([\![e_1]\!]_{expr}(p, \gamma), [\![e_2]\!]_{expr}(p, \gamma')) \\ \qquad\qquad \text{where } p, \gamma \vdash e_1 : c \parallel c' \parallel \phi, \text{ and } \gamma' = \gamma[c' \; \mathtt{this}]\end{array}$$

Object creation: Creation of instances of a non-state class c only requires invocation of the default constructor of c. If c is a state class, then two objects must be created: the implementor i (created by invoking the default constructor of c), and the wrapper w (created by invoking the proper constructor of class $\mathcal{R}(p, c)$, that is, the wrapper class of c). The implementor is passed as parameter to the constructor of the wrapper so that fields of w and i can be properly initialized to satisfy the equations $w.\mathtt{implementor} = i$ and $i.\mathtt{trueThis} = w$. The term $\mathcal{R}(p, c)$ denotes the least superclass of c which is not a state class: If c is a state class, then $\mathcal{R}(p, c)$ is its unique root superclass, otherwise $\mathcal{R}(p, c) = c$.

$$[\![\mathtt{new}\ c()]\!]_{expr}(p, \gamma) \stackrel{\Delta}{=} \begin{cases} \mathtt{new}\ \mathcal{R}(p, c)(\mathtt{new}\ c()) & \text{if } p \vdash c \diamond_s \\ \mathtt{new}\ c() & \text{otherwise} \end{cases}$$

This: The expression `this` is translated into `trueThis` because `this` could denote the implementor object i, rather than the wrapper w. Furthermore, the actual implementor of w may have changed because of re-classification, therefore `this` may denote an obsolete implementor. Because `trueThis` has static type `FickleObject`, in order to preserve types, the translation also needs to downcast to the root superclass of the type of `this`[4]. Note that since a state class c cannot be used as a type, the translation is statically correct also when `this` is passed as a parameter or assigned to a field.

$$[\![\mathtt{this}]\!]_{expr}(p, \gamma) \stackrel{\Delta}{=} (\mathcal{R}(p, \gamma(\mathtt{this})))\ \mathtt{trueThis}$$

5 Properties of the translation

In this section we formalize the properties of the translation previously mentioned. For lack of space we only sketch some proofs which will be detailed in a future extended version of this paper.

[4] Note that this downcasting is only necessary when `this` is used for parameter passing or assignments, and is unnecessary when `this` is used in method calls or field selection. This is so because in the latter cases field `implementor` of the object denoted by `trueThis` must be selected and `implementor` is declared in the type of `trueThis`. But, as already stated, we do not consider such optimization issues.

Preservation of static correctness

Theorem 1. *For any $\mathcal{F}ickle$ program p, if p is well-typed (in Fickle), then $[\![p]\!]_{prog}$ is well-typed (in Java).*

In order to be proved, the claim of the theorem must be extended to all subterms of p and, hence, to all typing judgments. The strengthened claim can be proved by induction on the typing rules. The claim concerning judgment for expressions is the most interesting, hence is stated below.

The translation preserves types up to state classes, in the following sense: if a *Fickle* expression e has type t w.r.t. a program p and an environment γ, and e is translated into a Java expression e' that has type t' w.r.t. $[\![p]\!]$ and γ, then $t = t'$, when t is not a state class, and t' is the root superclass of t, when t is a state class. For the Java fragment obtained from the translation we can use the *Fickle* type system, so that for any well-typed Java expression e we can derive judgments of the form $p, \gamma \vdash e \ : \ t \parallel \gamma(\texttt{this}) \parallel \emptyset$, where t is the type of e. The fact that the type of `this` remains the same, and the set of effects is empty indicates that e contains no re-classifications.

The claim for expressions can be formalized as follows:

Lemma 1. *For any $\mathcal{F}ickle$ expression e, program p, environment γ, if*

- $p, \gamma \vdash e \ : \ t \parallel c \parallel \phi,$ *and*
- $[\![e]\!]_{expr}(p, \gamma) = e',$ *and*
- $[\![p]\!]_{prog} = p',$

then

- $p', \gamma \vdash e' \ : \ \mathcal{R}(p, t) \parallel \gamma(\texttt{this}) \parallel \emptyset.$

Preservation of dynamic semantics We now show that the semantics of expressions is preserved by the translation. The semantics of the language *Fickle* we consider is the one introduced in [4]. Such semantics rewrites pairs of expressions and stores into pairs of values (or the exception `nullPntrExc`, indicating a reference to a null object), and stores. *Values*, denoted by $\mathtt{v}$, are either booleans, or integers, or addresses, denoted by ι. Stores map the unique parameter[5] `x` and the receiver `this` to values and addresses to objects. *Objects* are mappings between fields and values tagged by the class they belong to: $[[\mathtt{f}_1 : \mathtt{v}_1, \ldots, \mathtt{f}_r : \mathtt{v}_r]]^{c}$. We use o as a metavariable for objects, and if `f` is a field of o, $o(\mathtt{f})$ is the value associated to `f` in o.

The rewriting, defined in the context of a given program p that provides the definition for the classes used in the expression, is defined by the judgment $e, \sigma \underset{p}{\leadsto} \mathtt{v}, \sigma'$. The syntax of *Fickle* and the one of the Java fragment considered here are slightly different from the language of [4]. In particular there is a distinction between statements and expressions and classes have constructors.

[5] Recall that, for simplicity, we assume that in *Fickle* syntax each method definition has a unique parameter denoted by `x`.

However, the definition of the semantics in [4] can be easily adapted to deal with these features. Note that the Java fragment contains also casting. However, we do not need rules for casting, since well-typing will insure that casting is applied to objects that already have the target type.

To state the semantic correctness result we introduce a relation between stores $p \vdash \sigma \approx \sigma'$ that expresses the fact that store σ' is the "translation" of store σ. That is, an object o of class c in σ corresponds univocally to an object o' in σ' that is an instance of the translation of the class c. Both the store σ and the store σ' are assumed to agree with the relative environments and programs. That is, they contain values which agree, w.r.t. typing, with their definitions (see [4] for the formal definition of $p, \gamma \vdash \sigma \diamond$).

Definition 1. *Let $p, \gamma \vdash \sigma \diamond$ and $[\![p]\!], \gamma \vdash \sigma' \diamond$. We say that* v' *in* σ' corresponds to v *in σ w.r.t. p, and write $p, \sigma, \sigma' \vdash \mathsf{v} \approx \mathsf{v}'$, if either of the following conditions hold:*

- $\mathsf{v} = \mathsf{v}' = \mathtt{true}$, *or* $\mathsf{v} = \mathsf{v}' = \mathtt{false}$, *or* $\mathsf{v} = \mathsf{v}' = n$ *(for some integer n), or* $\mathsf{v} = \mathsf{v}' = \mathtt{null}$, *or*
- $\mathsf{v} = \iota$, $\mathsf{v}' = \iota'$, $\sigma(\iota) = [[\mathtt{f}_1 : \mathsf{v}_1, \ldots, \mathtt{f}_r : \mathsf{v}_r]]^{c}$,
  $\sigma'(\iota') = [[\mathtt{f}_1 : \mathsf{v}'_1, \ldots, \mathtt{f}_q : \mathsf{v}'_q, \mathtt{impl} : \iota'', \mathtt{trueThis} : \iota']]^{\mathcal{R}(p,c)}$, *$(q \leq r)$ and*
  $\sigma'(\iota'') = [[\mathtt{f}_1 : \mathsf{v}''_1, \ldots, \mathtt{f}_r : \mathsf{v}''_r, \mathtt{impl} : \iota'', \mathtt{trueThis} : \iota']]^{c}$, *and*
  for all i, $1 \leq i \leq r$, $p, \sigma, \sigma' \vdash \mathsf{v}_i \approx \mathsf{v}''_i$, and
  if c is not a state class, then $\iota' = \iota''$.

Note that if c is not a state class, then $\mathcal{R}(p, c) = c$, and so $q = r$. With this notion of correspondence between values we can define a correspondence between stores.

Definition 2. *Let $p, \gamma \vdash \sigma \diamond$ and $[\![p]\!], \gamma \vdash \sigma' \diamond$. We say that store σ'* corresponds to *σ w.r.t. p, and write $p \vdash \sigma \approx \sigma'$, if*

1. *$p, \sigma, \sigma' \vdash \sigma(\mathtt{x}) \approx \sigma'(\mathtt{x})$,*
2. *$p, \sigma, \sigma' \vdash \sigma(\mathtt{this}) \approx (\sigma'(\mathtt{this}))(\mathtt{trueThis})$, and*
3. *for all ι if $\sigma(\iota)$ is defined there is a unique ι' such that $p, \sigma, \sigma' \vdash \iota \approx \iota'$, and*
4. *for all ι' if $\sigma'(\iota')$ is defined there is a unique ι such that $p, \sigma, \sigma' \vdash \iota \approx (\sigma'(\iota'))(\mathtt{trueThis})$.*

The last two conditions of the previous definition assert that there is an injection between the set of addresses defined in σ and the set of addresses defined in σ'.

Theorem 2. *For a well-typed expression e, stores σ_0 and σ_1 such that $p, \gamma \vdash \sigma_0 \diamond$, $[\![p]\!], \gamma \vdash \sigma_1 \diamond$ and $p \vdash \sigma_0 \approx \sigma_1$,*

$$e, \sigma_0 \underset{p}{\rightsquigarrow} \mathsf{v}, \sigma'_0 \qquad \textit{if and only if} \qquad [\![e]\!], \sigma_1 \underset{[\![p]\!]}{\rightsquigarrow} \mathsf{v}', \sigma'_1$$

where $p \vdash \sigma'_0 \approx \sigma'_1$ and $p, \sigma, \sigma' \vdash \mathsf{v} \approx \mathsf{v}'$

The proof is by induction on the derivation of $e, \sigma \underset{p}{\rightsquigarrow} \mathsf{v}, \sigma'$. The proof that, in case of field selection and method call, the right method is selected relies on the following fact. If $p \vdash \sigma \approx \sigma'$, then: for all ι and c, $\sigma(\iota) = [[\cdots]]^{c}$ implies $\sigma'(\sigma'(\iota')(\mathtt{impl})) = [[\cdots]]^{c}$, where $p, \sigma, \sigma' \vdash \iota \approx \iota'$.

Support for separate compilation For any $\mathcal{F}ickle$ program p, let $classes(p)$ denote the set of all classes defined in p, and, for each class c in $classes(p)$, $dep_p(c)$ the set of all superclasses of c and of all classes (either directly or indirectly) used by c (for reasons of space we omit the formal definitions). The following claim states that a $\mathcal{F}ickle$ class declaration can be successfully translated in a $\mathcal{F}ickle$ program p whenever the set of dependencies of c is contained in p, exactly as happens for Java compilation.

Theorem 3. *For any well-formed $\mathcal{F}ickle$ program p and class declaration cld in p, if $dep_p(class(cld)) \subseteq classes(p)$, then $[\![cld]\!]_{cld}(p)$ is well-defined.*

Let *strip* be the function on $\mathcal{F}ickle$ programs defined as follows:

$$
\begin{array}{l}
strip(cld_1 \ldots cld_n) = strip(cld_1) \ldots strip(cld_n) \\
strip([\texttt{root} \mid \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\{field^*\ \ meth^*\}) = \\
\qquad [\texttt{root} \mid \texttt{state}]\ \texttt{class}\ c\ \texttt{extends}\ c'\{field^*\ \ strip(meth^*)\} \\
strip(meth_1 \ldots meth_n) = strip(meth_1) \ldots strip(meth_n) \\
strip(t\ m(t'\ x)\phi\{sl\ \texttt{return}\ e;\}) = t\ m(t'\ x)\phi\{\texttt{return}\ \mathsf{v}(t);\} \\
\mathsf{v}(t) = \begin{cases} \texttt{false} & \text{if } t = \texttt{boolean} \\ 0 & \text{if } t = \texttt{int} \\ \texttt{null} & \text{otherwise} \end{cases}
\end{array}
$$

The following theorem states that translation of a $\mathcal{F}ickle$ class c depends only on the body of c and the type information of all other classes, namely, class kind, parent class, method headers and field declarations. This information is stored in a regular Java class file[6], therefore the translation of c can be successfully carried out also when only the binary files of the other classes are available[7].

Theorem 4. *For any $\mathcal{F}ickle$ program p and $\mathcal{F}ickle$ class declaration cld_1, if $[\![cld_1]\!]_{cld}(p) = cld_2$, then $[\![cld_1]\!]_{cld}(strip(p)) = cld_2$.*

6 Conclusion

We have defined a translation from $\mathcal{F}ickle$ (a Java-like language supporting dynamic object re-classification) into plain Java, and proved that this translation well-behaves in the sense that it preserves static and dynamic semantics. This is a nice theoretical result, strengthened by the fact that, in order to ensure these properties, we were able to identify some invariants which turned out to be a very useful guide to the translation.

Our concerns are not only theoretical, but we are interested in investigating the possibility of implementing an extension of Java with re-classification. From this point of view, our translation is a good basis since it exhibits the following additional properties:

[6] Except for the kinds `root` and `state`, but class files format can be easily extended for storing this new piece of information.

[7] Note that this property does not depend on Java support for reflection.

– it is fully compatible with Java separate compilation, since each *Fickle* class can be translated without having other class bodies, hence in principle only having other classes in binary form;
– dependencies among classes are exactly those of standard Java compilation, in the sense that a *Fickle* class can be translated only if type information on all the ancestor and used classes is available.

Our translation is similar both in the structure of classes and in their behavior to the state pattern, see [5]. The wrapper class corresponds to the context class (of the pattern) and the implementation to the state class. Access to members require a level of indirection, as in the state pattern. So from the point of view of efficiency our implementation of reclassification performs as well as the state pattern. On the other side our translation maintains the structure of the original hierarchy, whereas the state pattern does not.

A prototype implementation largely based on the translation described in this paper has already been developed [2].[8] However, the work presented here is only a first step towards a working extension of Java with dynamic object re-classification. On one side, an extension of full Java should take into account other Java features (like constructors, access modifiers, abstract classes, interfaces, overloading and casting) which, though in principle orthogonal to re-classification, should be carefully analyzed in order to be sure that the interaction behaves correctly. On the other side, as mentioned above, an extended compiler should be able to work even in a context where only binary files are available, while our prototype implementation works on source files.

Finally, an alternative direction for the implementation of *Fickle* (or, more generally, of an object-oriented language supporting dynamic re-classification of objects) could be in a direct way, through manipulation of the object layout or the object look-up tables.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. In *ECOOP'00*, volume 1850 of *LNCS*, pages 154–178. Springer, 2000.
2. Christopher Anderson. Implementing Fickle, Imperial College, final year thesis - to appear, June 2001.
3. C. Chambers. Predicate Classes. In *ECOOP'93*, volume 707 of *LNCS*, pages 268–296. Springer, 1993.
4. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In J. L. Knudsen, editor, *ECOOP'01*, number 2072 in LNCS, pages 130–149. Springer, 2001. Also available in: Electronic proceedings of FOOL8 (http://www.cs.williams.edu/ kim/FOOL/).
5. R. Johnson E.Gamma, R. Elm and J. Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[8] The prototype is written in Java. Future releases might be written in (extended) *Fickle*.

6. M. D. Ernst, C. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *ECOOP'98*, volume 1445 of *LNCS*, pages 186–211. Springer, 1998.
7. M. Serrano. Wide Classes. In *ECOOP'99*, volume 1628 of *LNCS*, pages 391–415. Springer, 1999.