Electronic Notes in Theoretical Computer Science 62 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume62.html 14 pages

A type preserving translation of *Fickle* into Java

(Extended Abstract)

D. Ancona [1,3,7] C. Anderson [4] F. Damiani [1,5,8]
S. Drossopoulou [2,4,9] P. Giannini [1,6,10] E. Zucca [1,3,11]

Abstract

We present a translation from *Fickle* (a Java-like language allowing objects that can change their class at run-time) into plain Java. The translation, which maps any *Fickle* class into a Java class, is driven by an invariant that relates a *Fickle* object to its Java counterpart. The translation, which is proven to preserve both the static and the dynamic semantics of the language, is an enhanced version of a previous proposal by the same authors.

1 Introduction

Dynamic object re-classification is a feature which allows an object to change its class 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 [3] is a Java-like language supporting dynamic object re-classification, aimed at illustrating features for object re-classification which could extend

[1] Partially supported by MURST Cofin'99 Project "Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi (TOSCA)"
[2] Partially supported by EPSRC (Grant Ref:GR/L 76709)
[3] Address: DISI, Università di Genova, Genova, Italy.
[4] Address: Imperial College, London, UK.
[5] Address: Dipartimento di Informatica, Università di Torino, Torino, Italy.
[6] Address: DISTA, Università del Piemonte Orientale, Alessandria, Italy.
[7] Email: davide@disi.unige.it
[8] Email: damiani@di.unito.it
[9] Email: scd@doc.ic.ac.uk
[10] Email: giannini@di.unito.it
[11] Email: zucca@disi.unige.it

an imperative, typed, class-based, object-oriented language. A distinguished feature of *Fickle*, with respect to other proposals for dynamic object reclassification (see, *e.g.*, [2,4,5]), is that it is type-safe, in the sense that any type correct program is guaranteed never to access non-existing fields or methods. *Fickle* is essentially 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 reclassification. In particular, a *Fickle* class which does not use these features is a Java class.

In this paper we consider the problem of implementing, starting from the *Fickle* design, a working extension with dynamic object re-classification of a real object-oriented language. In particular, we show that a Java environment could be easily and naturally extended in such a way to handle standard Java and *Fickle* classes together.

In order to show this, we define a translation from *Fickle* into plain Java. The translation is proved to preserve static and dynamic semantics (that is, well-formed *Fickle* 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 *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. This is so because type information needed by the translation can be retrieved from type information stored in binary files.

In comparison with the previous version by the same authors [1], the translation presented here is simpler and furthermore preserves types.

The paper is organized as follows: In Section 2 we introduce *Fickle*. In Section 3 we describe the translation. In Section 4 we state the properties of the translation (preservation of static and dynamic semantics) [12] and illustrate the compatibility of the translation with Java separate compilation. In the Conclusion we summarize the relevance of this work and illustrate the advantages with respect to the translation described in [1].

2 The language *Fickle*

This section is not intended to be an exhaustive presentation of the language. We refer to [3] for a complete definition. The syntax of the language is specified in Fig. 1. We refer to [3] for the definition of the static semantics of *Fickle* (the type system of *Fickle* can be easily adapted to the subset of Java serving as target for the translation) and of some auxiliary functions used in the sequel.

In *Fickle* class definitions may be preceded by the keyword `state` or `root`

[12] Proofs will be provided in a forthcoming extended paper.

p ::= $class^*$

$class$ ::= [`root` | `state`] `class` c `extends` $c'\{field^*\ meth^*\}$

$field$ ::= $t\ f$

$meth$ ::= $t\ m(t'\ x)[\phi]\{sl$ `return` $e;\}$

t ::= `boolean` | `int` | c

ϕ ::= $\{c^+\}$

sl ::= s^*

s ::= $\{sl\}$ | `if` $(e)\ s_1$ `else` s_2 | $se;$ | `this!!`$c;$

se ::= $var = e$ | $e_1.m(e_2)$ | `new` $c()$

e ::= se | $sval$ | var | `this`

var ::= `x` | $e.f$

$sval$ ::= `true` | `false` | `null` | n

Fig. 1. Syntax of *Fickle*

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.
  We require state classes to extend either root 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`.
  We require root classes to extend only *plain*, *i.e.*, neither root nor state, classes

For a class c of a program p, $\mathcal{R}(p, c)$ denotes the *root superclass of* c if c is a state class, and c otherwise.

Objects of a plain class `C` behave like regular Java objects, that is, are never re-classified. However, since state classes are subclasses of plain 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 either state or root class `C` are created in the usual way by the expression `new C()`.

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.

Example 2.1 The following *Fickle* program defines the classes[13] `P`, `R`, `S1`, and `S2`.

```
class P extends Object{
 int f1;
 int m1(){R}{this.f1=1; return this.m2();}
 int m2(){R}{return this.f1;}
}

root class R extends P{ }

state class S1 extends R{
 int m2(){R}{this!!S2; this.f2=this.f1; return this.f2;}
 static void main(String[] args){System.out.println(new S1().m1());} }

state class S2 extends R{
 int f2;
 int m1(){return 3;}
 int m2(){R}{this.f2=1;return this.f1+this.f2;}
}
```

Consider the program in Example 2.1. Re-classifications are caused either directly by re-classification statements, like `this!!S2` in body of method `m2` of class `S1`, or indirectly by method calls, like `new S1().m1()` in the body of `main` which, in turn, causes the invocation of method `m2` of class `S1`. At the start of method `m2` of class `S1` the receiver is an object of class `S1`, therefore it has only the field `f1`, while it does not have the field `f2`. After execution of `this!!S2` the receiver is of class `S2`, the field `f1` retains its value while the field `f2` is now available.

Variables (that is, fields and parameters, since, for simplicity, *Fickle* does not have local variables) and return values of methods must be declared with 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. The only expressions whose type can be a state class are creation expression (`new C()`) and `this`; moreover, the type of `this` *may change*.

Annotations like {`R`} before 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 with no effects, like `m1` in class `S2`, do not cause any re-classification. Methods annotated by non-empty effects, like `m2` in class `S1`, may re-classify objects of (a subclass of) a class in their effect (in the example, of `R`).

A method annotated with effects can be overridden only by methods an-

[13] The class `S1` contains the method `main`. For simplicity, we have omitted the method `main` from both the *Fickle* syntax (in Fig. 1) and the formal definition of the translation (in Sections 3.2–3.5).

notated with the same or less effects [14] .

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. Typing an expression (or statement) d in the context of class declarations in program p and of type assumptions for parameters in environment γ

$$p, \gamma \vdash d : t \parallel c \parallel \phi$$

involves three components: t is the type of the value returned by the evaluation of d (if d is an expression) or `void` (if d is a statement), c is the type of `this` after the evaluation of d, and ϕ conservatively estimates the re-classification effect of the evaluation of d on objects. (See [3] for the typing rules.)

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 The translation of *Fickle* into Java

In this section we give a description of the translation. We first give an informal overview of the encoding of objects (Section 3.1), and then present the formal definition (Sections 3.2–3.5).

3.1 Encoding of 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 $<id, imp>$ of objects; we call *id* the *identity object of imp* and *imp* the *implementor object of id*. Roughly speaking, *id* provides the identity of o, and *imp* the behavior of o, so that any re-classification of o changes *imp* but not *id* and method invocations are resolved by *imp*. Hence, two implementors paired with the same identity represent the same object at different execution stages.

An object o which is not an instance of a state class does not need to be encoded in principle; however, for uniformity, the same kind of encoding described above is adopted also in this case, so that during the execution of a translated program there will be *exactly an identity object for any Fickle object*. Note that, while there could be more than one implementor for *Fickle* objects of state class, say $<id, imp>$ and $<id, imp'>$, the converse cannot be true: if $<id, imp>$ and $<id', imp>$ are translation of *Fickle* objects then $id = id'$. Re-classification of objects can be exemplified by the diagram in Fig. 2. Classes are translated according to the following two rules:

[14] 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.

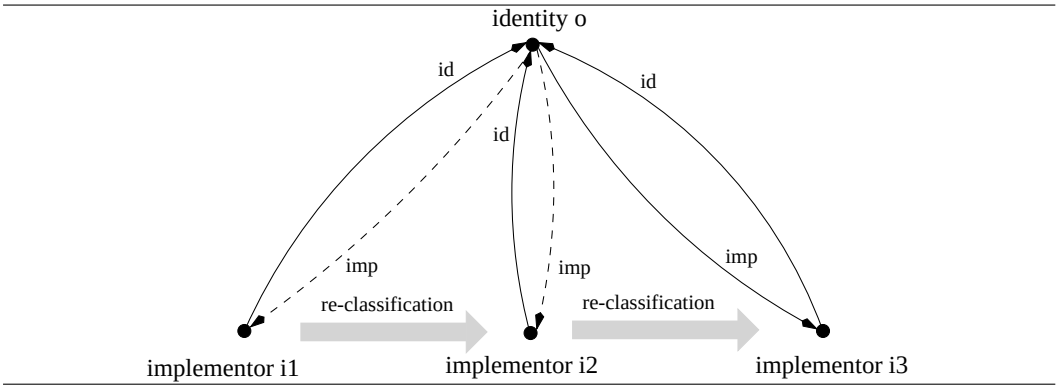

Fig. 2. Re-classification of objects

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

We illustrate the above in terms of the classes in Example 2.1. Let `s` have static type `R`. After the instruction

```
s=new S1();
```

the *Fickle* object referred by `s` is encoded in the translation, as sketched in Fig. 3, by the two Java objects $o1$ and $o2$ in which the field `imp` of $o2$ points to $o1$ (the dotted line). The variable `s` refers a Java object $o1$ of class `S1` with two

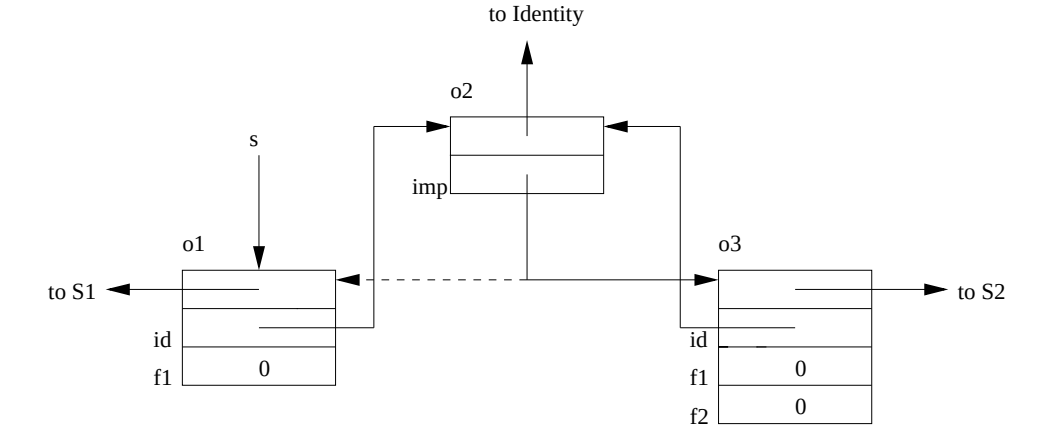

Fig. 3. Encoding of the *Fickle* object referred by `s`

fields: `f1` inherited from `P`, and `id` of type `Identity` and inherited from class `FickleObject` (see Fig. 4). The field `id` and `imp` are used in the translation for recovering the identity and the implementor of an object, respectively. In this case the field `id` points to an object $o2$ of class `Identity` that contains only a field `imp` that refers to the current implementor of the object (in this particular case the object $o1$ referred by `s` itself). After the re-classification `this!!S2` the Java object $o3$ of class `S2` is created and the field `imp` of the

```
class Identity extends Object{
 FickleObject imp;
 Identity(FickleObject theImp){this.imp=theImp;}
}

class FickleObject extends Object{
 Identity id;
 FickleObject(){                    // creates instances
  id=new Identity(this);
 }
 FickleObject(FickleObject oldImp){ // re-classifies objects
  id=oldImp.id;
  id.imp=this;
 }
}
```

Fig. 4. The classes `Identity` and `FickleObject`

identity $o2$ points to the new object $o3$. Note that the new implementor for the *Fickle* object referred by `s` can be recovered from the previous implementor $o1$ by accessing its `id` field denoting the identity object $o2$ and, then, by selecting field `imp` of $o2$.

3.2 Translation of programs

The translation of a *Fickle* program p consists of the translation of all classes declared in p. Since the translation of statements and expressions depends on their types, the program p is passed as parameter to the translation function for classes.

$$[\![p]\!]_{prog} \triangleq [\![class_1]\!]_{class}(p) \ldots [\![class_n]\!]_{class}(p), \qquad \text{where } p = class_1 \ldots class_n$$

3.3 Translation of classes

As previously said, each translated class extends class `FickleObject`. The definition of such a class along with the definition of the class `Identity` is given in Fig. 4.

The constructor `FickleObject()` is invoked whenever a new instance of a *Fickle* class is created and initializes the field `id` to a new identity.

On the other hand, constructor `FickleObject(FickleObject oldImp)` is invoked whenever an object is re-classified. An object o which needs to be re-classified to a state class c (recall that in the translation every class except for `Identity` is subclass of `FickleObject`), and which is encoded by the pair $<id, imp>$, is transformed into $<id, imp'>$, where imp' denotes the new implementor of class c (provided by a proper constructor of c; see definition below). The argument of the constructor denotes the old implementor imp, from which the identity id can be recovered, whereas imp' is denoted by `this`.

Fields are initialized so that the identity and the new implementor point to each other.

Each *Fickle* class c is translated into a single Java class containing the translation of all field and method declarations of c and a pair of constructors, used for creating instances and for re-classifying objects, respectively.

The translation of fields and methods is independent of the kind of class. However, the constructor for re-classifying an object in state classes is different from those defined in the other kinds of classes.

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

where $name(c') = \texttt{FickleObject}$ if $c' = \texttt{Object}$, and $name(c') = c'$ otherwise.

$$
\begin{array}{l}
[\![\texttt{state class } c \texttt{ extends } c'\{field_1 \ldots field_m\ meth_1 \ldots meth_n\}]\!]_{class}(p) \triangleq \\
\texttt{class } c \texttt{ extends } c'\{\ [\![field_1]\!]_{field}(c) \ldots [\![field_m]\!]_{field}(c) \\
\qquad\qquad [\![meth_1]\!]_{meth}(p,c) \ldots [\![meth_n]\!]_{meth}(p,c) \\
\qquad\qquad c()\{\} \\
\qquad\qquad c(\mathcal{R}(p,c)\ \texttt{oldImp})\{\texttt{super(oldImp);}\} \\
\qquad\qquad \}
\end{array}
$$

More precisely, the constructor $c(c\ \texttt{oldImp})$ for re-classification defined in both plain and root classes, after invoking the corresponding constructor in the superclass, copies all the fields of the old implementor (denoted by `oldImp`) declared in c in the corresponding fields of the new implementor (denoted by `this`). This step is not performed in the corresponding constructor $c(\mathcal{R}(p,c)\ \texttt{oldImp})$ in state classes since, according to the *Fickle* semantics, only the fields of the root superclass are preserved by re-classification.

3.4 Translation of fields

Translation of each field f comes equipped with a static method `to`f used for translating the assignments of a value v to a field f of an object o (see Section 3.7.3 below), since the implementor of the object o can be correctly selected only after evaluating v.

$$[\![t\ f;]\!]_{field}(c) \triangleq$$
$$t\ f;$$
$$\texttt{static}\ t\ \texttt{to}f(\texttt{FickleObject anImp}, t\ \texttt{x})\{\texttt{return}\ ((c)\ \texttt{anImp.id.imp}).f = \texttt{x};\}$$

3.5 Translation of 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 defining the type of the parameters and of `this` must be passed as argument to the corresponding translation functions.

$$[\![t\ m(t'\ \texttt{x})\phi\{sl\ \texttt{return}\ e;\}]\!]_{meth}(p,c) \triangleq$$
$$t\ m(t'\ \texttt{x})\{[\![sl]\!]_{stmts}(p,\gamma)\ \texttt{return}\ [\![e]\!]_{expr}(p,\gamma');\}$$
$$\texttt{static}\ t\ \texttt{call}m(\texttt{FickleObject anImp}, t'\ \texttt{x})\{\texttt{return}\ ((c)\ \texttt{anImp.id.imp}).m(\texttt{x});\}$$
$$\text{where } \gamma = t'\ \texttt{x}, c\ \texttt{this}, \quad \gamma' = t'\ \texttt{x}, c'\ \texttt{this}, \quad \text{and } p,\gamma \vdash sl : \texttt{void} \parallel c' \parallel \_.$$

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 `call`m used for translating invocations of m on receiver o and with argument `x` (see Section 3.7.3 below); indeed, the implementor of o can be correctly selected only after evaluating the argument x.

3.6 Translation of 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.

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

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

3.7 Translation of expressions

Types of expressions are preserved under the translation. This is formalized in Sect. 4.

3.7.1 Values, assignment to variables and object creation

The translation is straightforward.

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

3.7.2 Parameter, `this`*, and field selection*

In our encoding, in order to access the current implementor of an object we have to select the implementor currently pointed to by the identity of the object. For instance, the parameter `x` cannot be simply translated in itself, since `x` may refer to an obsolete implementor. Note that this problem does not occur for parameters and fields of type `int` and `boolean`.

$$[\![\mathtt{x}]\!]_{expr}(p,\gamma) \triangleq \begin{cases} ((t)\ \mathtt{x.id.imp}), & \text{if } t \text{ is a class} \\ \mathtt{x}, & \text{otherwise} \end{cases}$$
$$[\![\mathtt{this}]\!]_{expr}(p,\gamma) \triangleq ((c)\ \mathtt{id.imp})$$
$$[\![e.f]\!]_{expr}(p,\gamma) \triangleq \begin{cases} ((t)\ [\![e]\!]_{expr}(p,\gamma).f.\mathtt{id.imp}), & \text{if } t \text{ is a class} \\ ([\![e]\!]_{expr}(p,\gamma).f, & \text{otherwise} \end{cases}$$
where $p,\gamma \vdash \mathtt{x} : t \parallel \_ \parallel \_$, $p,\gamma \vdash \mathtt{this} : c \parallel \_ \parallel \_$, and $p,\gamma \vdash e.f : t \parallel \_ \parallel \_$.

Downcasting is needed after selection because field `imp` has type `FickleObject`.

3.7.3 Field assignment and method call

Field f (or method m) of the object denoted by the translation of e_1 is accessed through the implementor of its identity. However, e_2 could re-classify the object, therefore the selection `id.imp` is correct only after evaluating the translation of e_2. This is achieved by invoking the auxiliary static methods associated with fields and methods.

$$[\![e_1.f = e_2]\!]_{expr}(p,\gamma) \triangleq ((c'''')c.\mathtt{to}f([\![e_1]\!]_{expr}(p,\gamma), [\![e_2]\!]_{expr}(p,\gamma')))$$
$$[\![e_1.m(e_2)]\!]_{expr}(p,\gamma) \triangleq c.\mathtt{call}m([\![e_1]\!]_{expr}(p,\gamma), [\![e_2]\!]_{expr}(p,\gamma'))$$
where $p,\gamma \vdash e_1 : c' \parallel c'' \parallel \_, \gamma' = \gamma[c''\ \mathtt{this}], p,\gamma' \vdash e_2 : c''' \parallel \_ \parallel \phi$ and $c = \phi @_p c'$

The class c on which the static methods must be invoked is determined by applying the effect $\phi = \{c_1, \ldots, c_n\}$ to the static type c' of the expression e_1:

$$\{c_1,\ldots,c_n\}@_p c' = \begin{cases} c_i & \text{if } \mathcal{R}(p,c') = c_i \text{ for some } i \in 1,\ldots,n \\ c' & \text{otherwise} \end{cases}$$

Indeed, if the execution of e_2 re-classifies the object denoted by e_1, then the field f and the method m must be searched in $\mathcal{R}(p,c')$ rather than c'.

Instead of using static methods, another possibility would be to introduce local variables in which to store intermediate results, but in this case we would obtain a statement from the translation of an expression.

Example 3.1 The program in Example 2.1 is translated as follows.

```
class P extends FickleObject{
 int f1;
 static int tof1 (FickleObject anImp, int x) {return ((P)anImp.id.imp).f1=x;}
 int m1(){P.tof1(((P)id.imp),1);return P.callm2(((P)id.imp));}
 static int callm1(FickleObject anImp){return ((P)anImp.id.imp).m1();}
 int m2(){return ((P)id.imp).f1;}
 static int callm2(FickleObject anImp){return ((P)anImp.id.imp).m2();}
 P(){}
 P(P oldImp){super(oldImp); f1=oldImp.f1;}
}

class R extends P{
 R(){}
 R(R oldImp){super(oldImp);}
}

class S1 extends R{
 int m2(){new S2(this); S2.tof2(((S2)id.imp),((S2)id.imp).f1); return ((S2)id.imp).f2;}
 static int callm2(FickleObject anImp){return ((S1)anImp.id.imp).m2();}
 static void main(String[] args){System.out.println(R.callm1(new S1()));}
 S1(){}
 S1(R oldImp){super(oldImp);}
}

class S2 extends R{
 int f2;
 static int tof2 (FickleObject anImp, int x) {return ((S2)anImp.id.imp).f2=x;}
 int m1(){return 3;}
 static int callm1(FickleObject anImp){return ((S2)anImp.id.imp).m1();}
 int m2(){S2.tof2(((S2)id.imp),1); return ((S2)id.imp).f1+((S2)id.imp).f2;}
 static int callm2(FickleObject anImp){return ((S2)anImp.id.imp).m2();}
 S2(){}
 S2(R oldImp){super(oldImp);}
}
```

4 Properties of the translation

In this section we formalize the previously mentioned properties of the translation.

4.1 Preservation of static correctness

Theorem 4.1 *For any Fickle 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 translation preserves types in the following sense: if a *Fickle* expression e has type t w.r.t. a program p and an environment γ, then e is translated into an expression e' that has type t w.r.t. $[\![p]\!]$ and γ.

4.2 Preservation of dynamic semantics

The semantics of the language *Fickle* we consider is the one introduced in [3]. 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 v, are either booleans, or integers, or addresses, denoted by ι. Stores map parameters and the receiver `this` to values and map addresses to objects. *Objects* are mappings between fields and values tagged by the class they belong to: $[\![\mathtt{f_1} : \mathsf{v_1}, \ldots, \mathtt{f_r} : \mathsf{v_r}]\!]^c$.

The judgment $e, \sigma \overset{\rightarrow}{\underset{p}{\sim}} \mathsf{v}, \sigma'$ means that the evaluation of e in the store σ w.r.t. p produces the value v and modifies the store to σ'.

To state the semantic correctness result we introduce a relation between values $p, \sigma, \sigma' \vdash \mathsf{v} \approx \mathsf{v}'$ that says that v' in σ' is the translation of v in σ w.r.t. p. For primitive values such relation is the identity and for addresses it means that the objects referred by the addresses are one the translation of the other. This relation induces a relation between stores $p \vdash \sigma \approx \sigma'$ that expresses the fact that store σ' is the "translation" of store σ, that is, any object o of class c in σ is related to an object o' in σ'.

Theorem 4.2 *For a well-typed program p, a well-typed expression e, stores σ_0 and σ_1 such that $p \vdash \sigma_0 \approx \sigma_1$,*

$$e, \sigma_0 \overset{\rightarrow}{\underset{p}{\sim}} \mathsf{v}, \sigma_0' \qquad \textit{implies} \qquad [\![e]\!], \sigma_1 \overset{\rightarrow}{\underset{[\![p]\!]}{\sim}} \mathsf{v}', \sigma_1'$$

where $p \vdash \sigma_0' \approx \sigma_1'$ and $p, \sigma_0', \sigma_1' \vdash \mathsf{v} \approx \mathsf{v}'$.

The proof is by induction on the derivation of $e, \sigma_0 \overset{\rightarrow}{\underset{p}{\sim}} \mathsf{v}, \sigma_0'$.

4.3 Support for separate compilation

For any *Fickle* 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 dependencies of c,

that is, all superclasses of c and all classes (either directly or indirectly) used by c (we omit the trivial formal definitions). Furthermore, let $strip_c$ be the function on *Fickle* programs defined as follows:

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

The following theorem states that translation of a *Fickle* class c depends only on the body of c and the type information (namely, class kind, parent class, method headers and field declarations) of all its dependencies. This information is stored in a regular Java class file,[15] therefore the translation of c can be successfully carried out also when only the binary files of the other (Fickle) classes are available.[16] Note that this means that the Fickle language supports separate compilation, but does not imply anything on the compatibility between Fickle and Java code.

Theorem 4.3 *For any Fickle program p and declaration cld_1 of Fickle class c, if $[\![cld_1]\!]_{cld}(p) = cld_2$, then $[\![cld_1]\!]_{cld}(strip_c(p)) = cld_2$.*

5 Conclusion

We have defined a translation from *Fickle* (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.

The translation improves on a previous one by the same authors introduced in [1]. In the translation in [1] the encoding of objects was a pair $<w, i>$ of objects, where w was the *wrapper object of* i and i the *implementor object of* w. To preserve the hierarchy of the original program the wrapper, to which the program variables would refer to, was of root class, producing the following

[15] Except for the kinds `root` and `state`, but class files format can be easily extended for storing this new piece of information.
[16] This property does not depend on Java support for reflection.

problems.

(i) Types were preserved up to state classes. That is, if a *Fickle* expression e has type t and t is not a state class, then its translation has the same type, otherwise it has type t' where t' is the root superclass of t.

(ii) There was duplication of the fields of the root and plain superclasses of a state class.

The current translation solves both problems, making the translation of classes more uniform (as we can see from the creation of objects). However, whereas in the previous translation an object o not of state class was encoded by $<o, o>$, so no extra objects was created, in this new translation such object is encoded by $<id, o>$, where id is an identity object.

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] Ancona, D., C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini and E. Zucca, *An effective translation of Fickle into Java (extended abstract)*, in: *ICTCS'01*, LNCS 2202 (2001), pp. 210–230.

[2] Chambers, C., *Predicate Classes*, in: *ECOOP'93*, LNCS **707** (1993), pp. 268–296.

[3] Drossopoulou, S., F. Damiani, M. Dezani-Ciancaglini and P. Giannini, *Fickle: Dynamic object re-classification*, in: *ECOOP'01*, LNCS **2072** (2001), pp. 130–149.

[4] Ernst, M. D., C. Kaplan and C. Chambers, *Predicate Dispatching: A Unified Theory of Dispatch*, in: *ECOOP'98*, LNCS **1445** (1998), pp. 186–211.

[5] Serrano, M., *Wide Classes*, in: *ECOOP'99*, LNCS **1628** (1999), pp. 391–415.