Coalgebraic Semantics of an Imperative Class Based Language*

Furio Honsell,[1] Marina Lenisa[1]

Dipartimento di Matematica e Informatica, Università di Udine
Via delle Scienze 206, 33100 Udine, Italy.
tel. +39 0432 558417, fax: +39 0432 558499
e-mail: honsell,lenisa@dimi.uniud.it.

Abstract. We study two *observational equivalences* of *Fickle* programs. Fickle is a class-based object oriented imperative language, which extends Java with object *re-classification*. The first is a *contextual* equivalence of expressions with respect to a given program. We provide an *adequate coalgebraic semantics* for it, which is *compositional* w.r.t. the operators of the language. The second observational equivalence is defined on programs implementing the same specification, given as an *abstract class*. We introduce a coalgebraic description of classes which gives a sound *coinduction principle* for this latter equivalence. To this end we need to extend the original coalgebraic approach of H.Reichel and B.Jacobs to deal with *binary methods*, i.e. methods which take more than one instance of the hosting class as argument. This coalgebraic description induces in particular a *coinductive observational equivalence* on *objects* of a program, where objects (states of a class) are taken to be equal when the action of methods on them yield the same *observations* and equivalent next states.

Introduction

In recent years, in the Global Computing Community, there has been growing interest in *class-based object oriented* languages. In [DDDG02], the imperative typed class-based language *Fickle* has been introduced, which extends *Java* with *re-classification*. Re-classification allows objects to change class membership dynamically, while retaining their identity. In [DDDG02], a type system for Fickle is presented, which is sound w.r.t. the operational semantics. In particular, even though objects may be re-classified across classes with different members, there will never be an attempt to access non-existing members.

In this paper, we carry out the study of Fickle, by focusing on *observational equivalences* of *expressions programs*, *classes*, and *objects*. The results in this paper apply also to Java and similar languages. We consider Fickle as a representative of typed imperative class-based object oriented languages.

* Research supported by the UE project IST-2001-33477 DART, and the MIUR Project COFIN 2001013518 Cometa.

A program in Fickle is a sequence of classes. Each class consists of a sequence of *fields* (instance variables) and a sequence of *methods*. For simplicitly, we assume that all fields are *private*, i.e. outside the hosting class they are accessibile only through the methods; while all methods are *public*.

The first *observational equivalence* which we consider is on expressions. This is a *contextual* equivalence, called $\approx_P$, which is given w.r.t. a fixed program `P`. The equivalence $\approx_P$ equates `main` methods which have the same behaviour in any context, w.r.t. `P`. The second equivalence that we consider, called $\simeq$, is defined on programs P_1, P_2 implementing the same *program specification* `P`. As a program specification we simply take a sequence of *abstract classes* with no fields and a list of method declarations. A program P_1 implements a program specification `P`, whenever the method declarations in P_1 correspond exactly to the method declarations in `P`. Two programs P_1, P_2 implementing the same *program specification* `P` are equated under the equivalence $\simeq$ if and only if all `main` methods behave in the same way w.r.t. P_1 and P_2.

In this paper, we study *adequate coalgebraic semantics* of Fickle programs w.r.t. the equivalences $\approx_P$ and $\simeq$.

Coalgebraic semantics originated with Aczel-Mendler, Rutten-Turi, for CCS-like languages,[Acz88,AM89,Acz93,RT94], and it was further generalized to λ-calculus, [HL95], higher-order imperative languages, [Len96], object-oriented languages in a functional setting, [Rei95,Jac96], π-calculus, [HLMP98].

The gist of the *coalgebraic semantics paradigm* (*final semantics*) is to view the *interpretation* function from *syntax* to *semantics* as a *final* mapping in a suitable category. To this end the semantics has to be construed as a *final coalgebra* for a suitable functor F and the syntax has to be cast in form of an F-coalgebra. This approach is driven by the operational semantics of the language, because it is the semantics which determines the structure of the functor F. This is dual to the syntax-driven approach of *algebraic semantics* (*initial, denotational semantics*), where syntax is construed as an initial F-algebra and the semantics is defined as an F-algebra. The main advantage of the coalgebraic semantics is that it induces a *behavioural equivalence* on programs, which can be characterized as a *coalgebraic bisimilarity*, i.e. as greatest coalgebraic bisimulation.

The relations between coalgebraic and algebraic semantics have been studied in the general categorical setting of *bialgebras* by [TP97]. Bialgebras combine an algebraic and a coalgebraic structure. When there is a connection between the two structures, i.e. in the case of λ-bialgebras of [TP97], then the final semantics coincides with the initial semantics, i.e. it is both compositional w.r.t. the operators on the algebra and the equivalence induced is a bisimilarity.

In this paper, we introduce a simple coalgebraic semantics of programs directly driven by the operational semantics of Fickle expressions given in [DDDG02]. The coalgebraic semantics is *adequate* w.r.t. the contextual equivalence $\approx_P$ and can be seen to be compositional w.r.t. language constructors.

Then, following [Rei95,Jac96], we provide an alternative coalgebraic description of Fickle programs, where classes are modeled as coalgebras, whose carriers represent the objects (states) of the classes. The coalgebra models the evolu-

tion of the objects under the action of methods. The interest of this coalgebraic model lies in the fact that it determines behavioural equivalences on objects and classes (programs), which can be characterized as *bisimilarity equivalences*. In particular, the coalgebraic model gives a notion of equivalence between class implementations of the same class specification. In [Jac97,Jac97,HHJT98], coalgebraic refinement of classes has been studied.

Howevr, in the original coalgebraic approach only a single class *in isolation* is considered and the setting is purely *functional*. Moreover, *binary methods*, i.e. methods which take another instance of the hosting class as argument, cannot be described, since they would produce a contravariant occurrence of the variable in the corresponding functor. Extensions of the coalgebraic paradigm to mixed functors have been considered in [Tew00], but they are rather complex and they cover only a restricted range of cases.

In this paper, we extend the approach of [Rei95,Jac96], in order to model, in an *imperative* setting, general *programs*, i.e. sequences of classes possibly related by inheritance, mutual definitions, etc. In order to account for the *store* we model the evolution of objects together with their *references*.

In order to deal with binary methods, we model methods as *graphs* instead of *functions*, thus turning a (contravariant) function space functor into a (covariant) relations functor.

Our coalgebraic description of programs yields naturally a notion of equivalence $\dot{\simeq}$ on programs satisfying the same specification, which is expressed in terms of coalgebraic bisimulations between "initial" objects. Finally, we compare the equivalence $\dot{\simeq}$ to the observational equivalence $\simeq$.

Synopsis.

In Section 1, we recall the syntax and the operational semantics of Fickle and we introduce the observational equivalences on programs, which we will study. In Section 2, we introduce an adequate compositional coalgebraic semantics of Fickle expressions. In Section 3, we study a coalgebraic description of programs in the style of [Rei95,Jac96], together with the induced behavioural equivalences. Final remarks and directions for future work appear in Section 4. In Appendix A, some preliminaries on coalgebraic semantics are presented.

Acknowledgement. The authors would like to thank M.Dezani for helpful discussions.

1 The Language Fickle

In this section, first we recall the syntax and the operational semantics of the language Fickle (see [DDDG02] for more details). Then we introduce the observational equivalences on programs which we will study.

1.1 Syntax

Fickle syntax is summarized in Table 1. A Fickle program is a collection of class definitions. A class definition may be preceded by the keyword **state** or

root. State classes describe the properties of an object while it satisfies some conditions; when it no longer satisfies these conditions, it can be explicitly re-classified to another (state or root). Root classes abstract over state classes. Any subclass of a state or a root class must be a state class. Objects of a state class `c` may be re-classified to a class `c'`, where `c'` must be a subclass of the uniquely defined root superclass of *c*. Objects of a non-state, non-root class `c` behave like Java objects, i.e. they are never re-classified. The type of fields may be either boolean or integer or a non-state class. Thus fields may point to objects which change class, but these changes do not affect their type. In contrast, the type of `this` and parameters may be a state or root class; these variables may also point to objects which change class, and these changes affect their type.

Objects are created with the expression **new** `c`, where `c` is any class. Re-classification expressions, `id`⇓`c`, set the class of `id` to `c`, where `c` must be a state or a root class.

Methods declarations have the shape:

$$\texttt{t m } (\texttt{t}_1\texttt{x}_1, \ldots, \texttt{t}_\texttt{q}\texttt{x}_\texttt{q})\{\texttt{c}_1, \ldots, \texttt{c}_\texttt{n}\}\{\texttt{ e }\}$$

where `t` is the result type, $\texttt{t}_1, \ldots, \texttt{t}_\texttt{q}$ are the types of the formal parameters $\texttt{x}_1, \ldots, \texttt{x}_\texttt{q}$ and `e` is the body. The list of root classes $\texttt{c}_1, \ldots, \texttt{c}_\texttt{n}$ are the effect, i.e. the root classes of all objects that may be re-classified by invocation of that method.

For simplicitly, we assume all fields in the classes to be *private*, i.e. to be accessible from outside the class only through the class methods. On the contrary, we take all methods in a class to be *public*.

progr	:= class*
class	:= [**root** | **state**] **class** c **extends** c { field* meth* }
field	:= type f
meth	:= type m (par*) eff { e }
type	:= **bool** | **int** | c
par	:= type x
eff	:= { c* }
expr	:= **if** e **then** e **else** e | var:=e | e;e | sVal **this** | var | **new** c | e.m(e*) | id⇓c
var	:= x | e.f
sVal	:= **true** | **false** | **null**
id	:= **this** | x

Table 1. Syntax of Fickle

Example 1 (Lists in Fickle, [Dro02]). The Fickle program below consists of three classes: the root class *List* (which is *abstract*, since it contains only a sequence of method declarations) together with two subclasses, *EmptyList* and *NonEmptyList*. This program uses the re-classification, e.g. in the method *insertFront* of the class *EmptyList*.

```
abstract root class List extends Objects{
abstract insertFront(int i){List};
abstract getFront(){List};
abstract setFront(int i){List};
abstract setLast(List x){ }; ...}

state class EmptyList extends List{
void insertFront(int i){List}{
this⇓NonEmptyList; contents:= i; next := new EmptyList; }
int getFront (){}{ throw new ListException; } ...}

state class NonEmptyList extends List{
int contents;
List next;
NonEmptyList insertFront(int i){}{
NonEmptyList second:= new NonEmptyList;
copyTo(second); contents:= i; next:=second; }
int getFront(){List}{
int result := contents; next.copyTo(this); return result;}
List copyTo(NonEmptyList x){ } {
x.contents := contents; x.next:=next; } ...}
```

1.2 Operational Semantics

The operational semantics is given in terms of a SOS "big-step" relation $\longrightarrow$, which rewrites pairs of expressions and stores w.r.t. to a program `P` into pairs of values, exceptions, or errors, and stores. The expression which is evaluated is meant to represent the special method `main` (external to `P`) from which the execution of the program starts. The signature of the rewriting relation is:

$$\longrightarrow : \ store \to expr \times store \to (val \cup dev) \times store$$

Stores map `this` to an address, variables of base type to values, variables of class type to addresses, and addresses to objects. Stores have to be *finite*, i.e. collections of mappings with finite domain. Notice in particular that, in the store, addresses point to objects, but *not* to other addresses. Thus in Fickle, as in Java, pointers are implicit, and there are no pointers to pointers. We denote stores with σ, addresses with ι, exceptions and errors with `dv`.

$$
\begin{aligned}
\mathit{store} &\triangleq (\{\mathit{this}\} \rightarrow_{\mathit{fin}} \mathit{addr}) \cup (\mathit{varid} \rightarrow_{\mathit{fin}} \mathit{val}) \cup (\mathit{addr} \rightarrow_{\mathit{fin}} \mathit{object}) \\
\mathit{val} &\triangleq \mathit{sVal} \cup \mathit{addr} \\
\mathit{dev} &\triangleq \{\mathit{nullPntrExc}, \mathit{stuckErr}\} \\
\mathit{object} &\triangleq \{[f_1 : v_1, \ldots, f_r : v_r]^c \mid \ f_1, \ldots, f_r \text{ field identifiers}, v_1, \ldots, v_r \in \mathit{val}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c \ \text{ class name}\}
\end{aligned}
$$

Before introducing the rewriting rules, we need to define some operations on objects and stores. For object $o \triangleq [f : 1 : v_1, \ldots, f_l : v_l \ldots f_r : v_r]^c$, store σ, value $\mathtt{v}$, address ι, identifier or address z, field identifier $\mathtt{f}$, we define:

- *field access*: $o(f) \triangleq \begin{cases} v_l & \text{if } f = f_l \text{ for some } l \in 1, \ldots, r, \\ \mathit{Udf} & \text{otherwise} \end{cases}$
- *object update*: $o[f \mapsto v] \mapsto [f : 1 : v_1, \ldots, f_l : v \ldots f_r : v_r]^c$, where $f_l = f$ for some $l \in 1, \ldots, r$,
- *store update*: $\sigma[z \mapsto v](z) = v, \quad \sigma[z \mapsto v](z') = \sigma(z')$ if $z' \neq z$.

We use the convention that $\sigma(\iota)(f) = \mathit{Udf}$, whenever $\sigma(\iota) = \mathit{Udf}$.

Tables 2, 3 and 4 list the rewrite rules of the operational semantics.

The evaluation of the expression **new** `c` in a store σ extends σ with a new *canonical address*.

In the rule for method call, $e_0.m(e_1, \ldots, e_n$ in Table 2, we use the the function $\mathcal{M}(P, c, m)$ that returns the definition of method `m` in class `c` going through the class hierarchy (see [DDDG02] for more details).

For re-classification expressions, $\mathtt{id} \Downarrow d$, we find the address of `id`, which points to an object of class `c`. We replace the original object by a new object of class `d`. We preserve the fields belonging to the root superclass of `c` and initialize the other fields of `d` according to their types. The term $\mathcal{R}(P, t)$, defined by

$$
\mathcal{R}(P, t) = \begin{cases} c & \text{if } t \text{ is a state class and } c \text{ is the root superclass of } t \\ t & \text{otherwise} , \end{cases}
$$

denotes the least superclass of `t` which is not a state class, if `t` is a class, and denotes `t` itself if `t` is not a class.

1.3 Observational Equivalences

Various notions of *observational equivalences* on Fickle programs are naturally induced by the operational semantics. First of all, one can define a *contextual equivalence* on `main` methods w.r.t. a given program `P`, by evaluating the expressions corresponding to the bodies of the `main` methods in any expression context `C[ ]`, and by observing the output value. A context is simply an expression with a hole. As observable values, we take values of base types, i.e. $\mathit{obsval} \triangleq \mathit{bool} \cup \mathit{int}$.

$$\frac{(e,\sigma) \longrightarrow_P (\text{true},\sigma'') \quad (e_1,\sigma'') \longrightarrow_P (v,\sigma')}{(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2,\sigma) \longrightarrow_P (v,\sigma')} \qquad \frac{(e,\sigma) \longrightarrow_P (\text{false},\sigma'') \quad (e_2,\sigma'') \longrightarrow_P (v,\sigma')}{(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2,\sigma) \longrightarrow_P (v,\sigma')}$$

$$\frac{\sigma(x) \neq \mathit{Udf} \quad (e,\sigma) \longrightarrow_P (v,\sigma')}{(x := e,\sigma) \longrightarrow_P (v,\sigma'[x \mapsto v])} \qquad \frac{(e,\sigma) \longrightarrow_P (\iota,\sigma'') \quad (e',\sigma'') \rightarrow_P (v,\sigma''') \quad \sigma'''(\iota)(f) \neq \mathit{Udf} \quad \sigma' \triangleq \sigma'''[\iota \mapsto \sigma'''(\iota)[f \mapsto v]]}{(e.f := e',\sigma) \longrightarrow_P (v,\sigma')}$$

$$\frac{(e_1,\sigma) \longrightarrow_P (v',\sigma'') \quad (e_2,\sigma'') \longrightarrow_P (v,\sigma')}{(e_1;e_2,\sigma) \rightarrow_P (v,\sigma')} \qquad \frac{(e,\sigma) \longrightarrow_P (\iota,\sigma') \quad \sigma'(\iota)(f) \neq \mathit{Udf}}{(e.f,\sigma) \rightarrow_P (\sigma'(\iota)(f),\sigma')}$$

$$\frac{\sigma(\mathit{id}) \neq \mathit{Udf}}{(\text{id},\sigma) \longrightarrow_P (\sigma(\mathit{id}),\sigma)} \qquad \frac{}{(v,\sigma) \longrightarrow_P (v,\sigma)}$$

$$\frac{\begin{array}{l}\mathcal{F}_S(P,c) = \{f_1,\ldots,f_r\} \\ v_l \text{ initial for } \mathcal{F}(P,c,f_l)\ (\forall l \in \{1,\ldots,r\}) \\ \iota \text{ is new in } \sigma\end{array}}{(\textbf{new } c,\sigma) \longrightarrow_P (\iota,\sigma[\iota \mapsto [f_1 : v_1,\ldots,f_r : v_r]^c])}$$

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0) \\ (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,n\}) \\ \sigma_n(\iota) = [\ldots]^c \\ \mathcal{M}(P,c,m) = t\ \ m(t_1x_1,\ldots,t_nx_n)\ \ \phi\ \ \{\ \ e\ \ \} \\ \sigma' = \sigma_n[\text{this} \mapsto \iota, x_1 \mapsto v_1,\ldots,x_n \mapsto v_n] \\ (e,\sigma') \longrightarrow_P (v,\sigma'')\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma') \longrightarrow_P (v,\sigma''[\text{this} \mapsto \sigma_n(\text{this}), x_1 \mapsto \sigma_n(x_1),\ldots,x_n \mapsto \sigma_n(x_n)])}$$

$$\frac{\begin{array}{l}\sigma(\text{id}) = \iota \\ \sigma(\iota) = [\ldots]^c \\ \mathcal{F}_S(P,\mathcal{R}(P,c)) = \{f_1,\ldots,f_r\} \\ v_l = \sigma(\iota)(f_l)\ (\forall l \in \{1,\ldots r\}) \\ \mathcal{F}_S(P,d) \setminus \{f_1,\ldots,f_r\} = \{f_{r+1},\ldots,f_{r+q}\} \\ v_l \text{ initial for } \mathcal{F}_S(P,d,f_l)\ (\forall l \in \{r+1,\ldots r+q\})\end{array}}{(\text{id} \Downarrow d,\sigma) \longrightarrow_P (\iota,\sigma[\iota \mapsto [f_1 : v_1,\ldots,f_{r+q} : v_{r+q}]^d)} \qquad \frac{(\text{id},\sigma) \longrightarrow_P (\text{null},\sigma')}{(\text{id} \Downarrow d,\sigma) \longrightarrow_P (\text{null},\sigma')}$$

Table 2. Operational Semantics: execution without exceptions and errors

$$\frac{(e,\sigma) \longrightarrow_P (\text{null},\sigma')}{\begin{array}{l}(e.f := e',\sigma) \longrightarrow_P (\text{nullPntrExc},\sigma') \\ (e.f,\sigma) \longrightarrow_P (\text{nullPntrExc},\sigma') \\ (e.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\text{nullPntrExc},\sigma')\end{array}}$$

$$\frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (v,\sigma') \\ v \neq \text{true and } v \neq \text{false}\end{array}}{(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2,\sigma) \longrightarrow_P (\text{stuckErr},\sigma')} \qquad \frac{\sigma(x) = \text{true or } \sigma(x) = \text{false}}{(x \Downarrow c,\sigma) \longrightarrow_P (\text{stuckErr},\sigma)}$$

$$\frac{\sigma(x) = \mathit{Udf}}{\begin{array}{l}(x,\sigma) \longrightarrow_P (\text{stuckErr},\sigma) \\ (x := e,\sigma) \longrightarrow_P (\text{stuckErr},\sigma) \\ (x \Downarrow c,\sigma) \longrightarrow_P (\text{stuckErr},\sigma)\end{array}} \qquad \frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (v,\sigma') \\ v \neq \text{null} \\ v \notin \text{addr}\end{array}}{\begin{array}{l}(e.f,\sigma) \longrightarrow_P (\text{stuckErr},\sigma') \\ (e.f := e',\sigma) \longrightarrow_P (\text{stuckErr},\sigma')\end{array}}$$

$$\frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\iota,\sigma') \\ \sigma'(\iota)(f) = \mathit{Udf}\end{array}}{(e.f,\sigma) \longrightarrow_P (\text{stuckErr},\sigma')} \qquad \frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\iota,\sigma'') \\ (e',\sigma'') \longrightarrow_P (v,\sigma') \\ \sigma'(\iota)(f) = \mathit{Udf}\end{array}}{(e.f := e',\sigma) \longrightarrow_P (\text{stuckErr},\sigma')}$$

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (v,\sigma_0) \\ v \neq \text{null} \\ v \notin \text{addr or } \sigma_0(v) = \mathit{Udf}\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\text{stuckErr},\sigma_0)} \qquad \frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0) \\ (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,n\}) \\ \sigma_n(\iota) = [\ldots]^c \\ \mathcal{M}(P,c,m) = \mathit{Udf}\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\text{stuckErr},\sigma_n)}$$

Table 3. Operational semantics: generation of exceptions and errors

$$\frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\mathrm{dv},\sigma') \text{ or}\\ ((e,\sigma) \longrightarrow_P (\mathrm{true},\sigma'') \text{ and } (e_1,\sigma'') \longrightarrow_P (\mathrm{dv},\sigma')) \text{ or}\\ ((e,\sigma) \longrightarrow_P (\mathrm{false},\sigma'') \text{ and } (e_2,\sigma'') \longrightarrow_P (\mathrm{dv},\sigma'))\end{array}}{(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2,\sigma) \longrightarrow_P (\mathrm{dv},\sigma')}$$

$$\frac{(e_1,\sigma) \longrightarrow_P (\mathrm{dv},\sigma') \text{ or } ((e_1,\sigma) \longrightarrow_P (v,\sigma'') \text{ and } (e_2,\sigma'') \longrightarrow_P (\mathrm{dv},\sigma'))}{(e_1;e_2,\sigma) \longrightarrow_P (\mathrm{dv},\sigma')}$$

$$\frac{(e,\sigma) \longrightarrow_P (\mathrm{dv},\sigma')}{\begin{array}{l}(x := e,\sigma) \longrightarrow_P (\mathrm{dv},\sigma')\\ (e.f,\sigma) \longrightarrow_P (\mathrm{dv},\sigma')\\ (e.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\mathrm{dv},\sigma')\\ (e.f := e',\sigma) \longrightarrow_P (\mathrm{dv},\sigma')\end{array}} \qquad \frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\iota,\sigma'')\\ (e',\sigma'') \longrightarrow_P (\mathrm{dv},\sigma')\end{array}}{(e.f := e',\sigma) \longrightarrow_P (\mathrm{dv},\sigma')}$$

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0)\\ (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,q\},\ q < n)\\ (e_{q+1},\sigma_q) \longrightarrow_P (\mathrm{dv},\sigma_{q=1})\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\mathrm{dv},\sigma_{q+1})}$$

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0)\\ (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,n\})\\ \sigma_n(\iota) = [\ldots]^c\\ \mathcal{M}(P,c,m) = t\ m(t_1x_1,\ldots,t_n : x_n)\ \phi\ \{\ e\ \}\\ \sigma' = \sigma_n[\mathrm{this} \mapsto \iota, x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]\\ (e,\sigma') \longrightarrow_P (\mathrm{dv},\sigma'')\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\mathrm{dv},\sigma''_{[}\mathrm{this} \mapsto \sigma_n(\mathrm{this}, x_1 \mapsto \sigma_n(x_1),\ldots,x_n \mapsto \sigma_n(x_n)])}$$

Table 4. Operational semantics: propagation of exceptions and errors

Definition 1 (Contextual Equivalence). *Let* $\approx_P \subseteq expr \times expr$ *be defined by*

$$e \approx_P e' \iff \forall C[\,]\ \forall \sigma\ \forall v \in obsval.\ (C[e], \sigma) \Downarrow_P v \ \Leftrightarrow\ (C[e'], \sigma) \Downarrow_P v\ .$$

The contextual equivalence $\approx_P$ on expressions `e,e'` induces an equivalence between a program `P` together with a `main` method whose body is the expression `e`, and the same program `P` together with a `main` method whose body is the expression `e'`.

In Section 2, we will provide an adequate compositional semantics of Fickle expressions w.r.t. the equivalence $\approx_P$.

In the definition of the observational equivalence $\approx_P$, the program `P` is fixed. This is a bit restrictive. An interesting issue is that of establishing equivalences between different programs P_1, P_2, which implement the same program specification. A simple notion of program specification can be taken to be a list of *abstract classes* with no fields and a sequence of method declarations. Then a program P_1 implements a program specification `P`, when the method declarations in P_1 correspond exactly to the method declarations in `P`. One could consider a more sophisticated notion of program specification, involving a first-order logic for expressing conditions on the fields. This would be useful for studying program refinement (see e.g. [ST97]). But, for our purposes, our simpler definition is sufficient.

Two programs P_1, P_2, implementing the same program specification `P`, can be taken to be equivalent, when for any possible `main` method, they evaluate to the same value:

Definition 2 (Program Equivalence). *Let* P_1, P_2 *implement the same program specification* `P`*. We define the equivalence* $\simeq$ *by:*

$$P_1 \simeq P_2 \iff \forall e\ \forall v \in obsval.\ (e, \emptyset) \Downarrow_{P_1} v \ \Leftrightarrow\ (e, \emptyset) \Downarrow_{P_2} v\ .$$

In Section 3, we will present a coalgebraic description of Fickle programs, yielding an equivalence between implementations of a specification, which can be characterized as a coalgebraic bisimilarity.

2 A Coalgebraic Semantics of Fickle Expressions

The coalgebraic semantics of expressions which we define is directly induced by the operational semantics. It is induced by a constant functor. However, the coalgebraic perspective is rather useful since it allows us to gain an interesting characterization of the equivalence induced by the semantics. Moreover, this semantics is compositional and adequate w.r.t. $\approx_P$.

One can endow the set of expressions with a structure of coalgebra induced by the operational semantics. Let us consider be the following (constant) functor $H : Set \to Set$:

$$HX \triangleq store \to ((addr + bool + int + dev) \times \widehat{store} + 1)\ ,$$

where $\widehat{store}$ denotes the set of *restricted stores*, obtained by restricting the domain of the function $addr \rightarrow_{fin} object$ to those addresses which are directly or indirectly images of an identifier. An address is indirectly associated to an identifier if it is associated to a field of a field of ... of an object. This definition is justified by the fact that, as noticed at the beginning of Subsection 1.2, addresses are not directly observable in the language, but only through identifiers.

Definition 3 (Coalgebraic Semantics). *i) Let $(expr, \alpha_e)$ be the H-coalgebra defined by*

$$\alpha_e(e) \triangleq \begin{cases} \sigma \mapsto (v, \sigma_1) & \text{if } (e,\sigma) \rightarrow_P (v, \widehat{\sigma}_1) \\ * & \text{otherwise} \end{cases}$$

where $\widehat{\sigma}_1 \in \widehat{store}$ denotes the restriction of the store σ_1.
ii) Let $\mathcal{M}_P : (expr, \alpha_e) \rightarrow (\Omega_H, \alpha_{\Omega_H})$ be the unique morphism into the final H-coalgebra $(\Omega_H, \alpha_{\Omega_H})$.

Proposition 1. *The coalgebraic semantics $\mathcal{M}_P$ induces the following bisimilarity equivalence:*

$$e \dot{\approx}_P e' \iff \forall\sigma\ \forall v.\ (e,\sigma) \rightarrow_P (v,\sigma_1)\ \wedge\ (e',\sigma) \rightarrow_P (v,\sigma_1')\ \wedge\ \widehat{\sigma}_1 = \widehat{\sigma}_1' \ .$$

One can easily check that:

Lemma 1.

$$e \dot{\approx}_P e' \iff \forall\sigma,\sigma',\sigma_1,\sigma_1'.\ \widehat{\sigma} = \widehat{\sigma}' \Rightarrow\ (e,\sigma) \rightarrow_P (v,\sigma_1) \wedge (e',\sigma') \rightarrow_P (v,\sigma_1') \wedge \widehat{\sigma}_1 = \widehat{\sigma}_1' \ .$$

Hence $\mathcal{M}_P$ can be equivalently defined as

$$\mathcal{M}_P : expr \rightarrow \widehat{store} \rightarrow ((addr + bool + int + dev) \times \widehat{store} + 1)$$

Moreover:

Lemma 2. *$\dot{\approx}_P$ is a congruence, i.e.: $e \dot{\approx}_P e' \implies \forall C[\,].\ C[e] \dot{\approx}_P C[e']$.*

Corollary 1 (Compositionality). *$\mathcal{M}_P$ is compositional.*

Theorem 1 (Adequacy). *For all expressions e, e',*

$$e \dot{\approx}_P e' \implies e \approx_P e' \ .$$

However, we conjecture that the semantics $\mathcal{M}_P$ is *not* fully abstract. Intuitively, there are cases in which the output store is affected, but not the output value.

3 Coalgebraic Description of Fickle Programs

In this section, we give a coalgebraic account of Fickle programs. Following [Rei95,Jac96], we model classes as coalgebras, where the carrier represents the objects (states) of the classes, and the coalgebra structure is determined by the operational semantics of the methods. The coalgebra structure captures the evolution of the objects under the action of methods.

This model will naturally induce a coinductive equivalence on objects of a program P. Moreover, program implementations P_1, P_2 of the same specification will be modeled by coalgebras for the same functor. Therefore, our coalgebraic description will induce also a notion of equivalence on program implementations, given in terms of coalgebraic bisimulations between "initial" objects.

In order to model the evolution of objects in an imperative setting, we need to account also for their references in the store. Moreover, objects, as they are defined in Section 1, possibly contain references to other objects in the fields. Thus we need to consider an enriched notion of object, which is *p*rima facie a finite set closed under referred objects. Formally, such notion of enriched objects w.r.t. a Fickle program P is defined as follows.

Definition 4. *i) Let* refobject *be defined by:*

$$(\mathit{refobject} \ni)\ o ::= \ (\iota, [f_1 : w_1, \ldots, f_r : w_r]^c) \ ,$$

where w_i is a value of base type, if f_i is of base type, an element of refobject, otherwise.
ii) Let $\overline{\mathit{refobject}}$ be the set of all pairs (o, O), where $o \in$ refobject, and $O \subseteq_{fin}$ refobject is a minimal coherent *set* closed *under referred object generated by o, i.e.*

- *(*closure*) $o \in O$, and, for all $o' \in O$, any object o'' referred by o' (i.e. o'' is associated to a field in o') is in O;*
- *(*coherence*) for all $o', o'' \in O$, if $\pi_1(o') = \pi_1(o'')$, then $\pi_2(o') = \pi_2(o'')$;*
- *(*minimality*) if $o' \in O$, then o' is (possibly indirectly) referred by o.*

In what follows, we simply denote by o an element (o, O) of $\overline{\mathit{refobject}}$.

Now, we endow the set $\overline{\mathit{refobject}}$ of enriched objects of P with a coalgebra structure for the functor induced by the methods as follows:

Definition 5. *Let $P \triangleq c_1, \ldots, c_n$, where $c_i \triangleq \{f_{i1}; \ldots f_{ih_i}; m_{i1}; \ldots; m_{ik_i}\}$.*
i) Let $F : Set \to Set$, be defined by

$$F \triangleq \Pi_{ij} F_{ij} \ ,$$

where $F_{ij} : Set \to Set$ is determined by the method $\mathtt{m}_{ij}$ $(t_1 x_1, \ \ldots, \ t_q x_q)$ $\{$ $c'_1, \ldots, c'_p\}\{e\}$ of the class c_i as follows:

$$F_{ij} X \triangleq [\![t_1]\!] \times \ldots \times [\![t_q]\!] \to (([\![t]\!] + \{nullPntrExc, stuckErr\}) \times X + 1) \ ,$$

where $[\![t_i]\!] = \begin{cases} bool & \text{if } t_i = bool \\ int & \text{if } t_i = int \\ X & \text{otherwise .} \end{cases}$

ii) $\alpha : \overline{refobject} \to F(\overline{refobject})$ *is defined by*

$$\alpha \triangleq \langle \alpha_{ij} \rangle_{ij} \ ,$$

where $\alpha_{ij} : \overline{refobject} \to F_{ij}(\overline{refobject})$ *is defined by*

$$\alpha_{ij}(o) \triangleq \boldsymbol{a} \mapsto \begin{cases} (v, \sigma_1[this]) & \text{if } (e, \sigma[this \mapsto o, \boldsymbol{x} \mapsto \boldsymbol{a}]) \longrightarrow_P (v, \sigma_1) \\ * & \text{otherwise} \ , \end{cases}$$

where, by abuse of notation, $\sigma[this \mapsto o, \boldsymbol{x} \mapsto \boldsymbol{a}]$ *denotes the store* σ *in which the object corresponding to the* refobject o *has been associated to the identifier* `this`*, and* σ *has been updated according to the* refobject o*. Similarly for* refobject *parameters.*

However, the functor F_{ij} (and hence F) is not guaranteed to be covariant. Namely, binary methods produce contravariant occurrences of X in the definition of F_{ij}. An example of a binary method is the method equal : $c \times c \to bool$, which takes another instance of the hosting class as argument. The second occurrence of c produces a contravariant occurrence of X in $FX \triangleq \ldots \times (\mathbf{X} \to (bool + dev) \times X) \times \ldots$. Therefore, the coalgebraic approach does not apply. We propose to turn contravariant occurrences in covariant by modeling *methods* as *graphs* instead of functions, i.e. in the case of the method *equal*, $FX \triangleq \ldots \times \mathcal{P}(X \times (bool + dev) \times X) \times \ldots$. This allows us to apply the standard coalgebraic approach, provided we move to a category where the functor F so defined admits a final coalgebra, e.g. the category *Class** of *classes* of *non-wellfounded sets* of [Acz88]. Then we can define the coalgebraic semantics of `P` as the unique morphism $[\![\ ]\!]_P : (\overline{refobject}, \alpha) \to (\Omega_F, \alpha_{\Omega_F})$ into the final F-coalgebra $(\Omega_F, \alpha_{\Omega_F})$. An interesting issue now arises, namely that of studying the equivalence induced by $[\![\ ]\!]_P$ on objects of `P`.

3.1 Coalgebraic equivalence on Objects

The equivalence induced by $[\![\ ]\!]_P$ on objects can be characterized as follows:

Theorem 2. *The coalgebraic semantics* $[\![\ ]\!]_P : (\overline{refobject}, \alpha) \to (\Omega_F, \alpha_{\Omega_F})$ *induces the following bisimilarity:*

$$o \sim_P o' \iff$$

$\forall$ *method* $m : e$ *in* P. $\forall \sigma \ \exists \sigma'$. $\sigma \sim_P \sigma' \ \wedge \ (e, \sigma[this \mapsto o]) \longrightarrow_P (v, \sigma_1) \wedge$
$(e, \sigma[this \mapsto o']) \longrightarrow_P (v, \sigma_1') \ \wedge \ \sigma_1[this] \sim_P \sigma_1'[this]$,

where $\sigma \sim_P \sigma'$ *is the extension to stores of* $\sim_P$*, i.e.* $\forall x : t, \ t$ *base type.* $\sigma(x) = \sigma'(x)$ *and* $\forall x : c.\ \sigma(x) \sim_P \sigma'(x)$*, where, by abuse of notation, by* $\sigma(x)$ *we denote the "extended" object in* $\overline{refobject}$ *associated to* x*.*

Notice the alternation of quantifiers $\forall\sigma\ \exists\sigma' \ldots$ in the above characterization of $\sim_P$, which is induced by the powerset. The equivalence $\sim_P$ is *covariant*, in the sense that it can be viewed as the greatest fixed point of the operator on relations naturally associated to it. Under suitable conditions on P, the "contravariance which is missing" in the functor F, can be recovered in the notion of bisimilarity, in the sense that $\sim_P$ can be alternatively characterized as an equivalence whose associated operator is not monotone:

Theorem 3 (Contravariancy of $\sim_P$). *If all the fields of the objects are observable (i.e. there are methods returning the value of the fields), then*

$$o \sim_P o' \iff$$

$$\forall\ \textit{method}\ m : e\ \textit{in}\ P.\ \forall\sigma \sim_P \sigma'.\ (e, \sigma[\textit{this} \mapsto o]) \longrightarrow_P (v, \sigma_1) \wedge (e, \sigma[\textit{this} \mapsto o']) \longrightarrow_P (v, \sigma_1') \ \wedge\ \sigma_1[\textit{this}] \sim_P \sigma_1'[\textit{this}]\ .$$

3.2 Coalgebraic Equivalence of Programs

The coalgebraic description of programs given above yields a natural notion of equivalence between program implementations P_1, P_2 of the same specification P. Namely, if P_1 and P_2 are implementations of the same specification P, then P_1, P_2 are described by coalgebras for the same functor F, $(\overline{\textit{refobject}}_{P_1}, \alpha_{P_1})$ and $(\overline{\textit{refobject}}_{P_2}, \alpha_{P_2})$, respectively.

Definition 6. *The programs P_1, P_2 implementing the same specification* P *are* coalgebraic equivalent*, $P_1 \dot{\simeq} P_2$, if, for any initial object (i.e. an object with the fields set to initial values) $o_1 \in \overline{\textit{refobject}}_{P_1}$ there is an initial object $o_2 \in \overline{\textit{refobject}}_{P_2}$ such that there exists a coalgebraic bisimulation $\mathcal{R} \subseteq \overline{\textit{refobject}}_{P_1} \times \overline{\textit{refobject}}_{P_2}$ such that $(o_1, o_2) \in \mathcal{R}$, and vice versa.*

An interesting question which remains to be addressed is what are the relations between the equivalence $\simeq$ of Subsection 1.3 and the coalgebraic equivalence $\dot{\simeq}$. We conjecture that $\dot{\simeq} \subseteq \simeq$.

4 Final Remarks and Directions for Future Work

In this paper we have considered various notions of observational equivalences on Fickle programs, and corresponding coalgebraic equivalences. In particular, we have defined a compositional (coalgebraic) semantics of expressions, which is adequate w.r.t. a contextual equivalence of programs. Moreover, we have extended the coalgebraic description of [Rei95,Jac96] to imperative programs, also capturing binary methods.

In the future, we plan to:

- use the coalgebraic semantics of expressions of Section 2 to give an alternative proof to type soundness of [DDDG02];

– extend the coalgebraic description of Fickle programs of Section 3 to bialgebras, modeling *method constructors* as algebra operations;
– study *coalgebraic program specification* and *refinement* (cfr. class specification of [Jac97,Jac97a]); which properties are preserved under coalgebraic refinement?
– study labeled transition systems for defining the operational semantics of Fickle and give corresponding coalgebraic semantics;
– extend the coalgebraic model to other advances OO features (mixin, modules, ...).

References

[Acz88] P.Aczel. *Non-well-founded sets*, CSLI Lecture Notes **14**, Stanford 1988.

[Acz93] P.Aczel. *Final Universes of Processes*, *MFPS'93*, Brookes et al. eds., LNCS **802**, 1993.

[AM89] P.Aczel, N.Mendler. A Final Coalgebra Theorem, in *Category Theory and Computer Science*, D.H.Pitt et al. eds., Springer LNCS **389**, 1989, 357–365.

[Dro02] S.Drossopoulou, Three Case studies in $Fickle_{II}$, Tech. rep., Imperial College. Available from `http://www.di.unito.it/~damiani/papers/dor.html`.

[DDDG02] S.Drossopoulou, F.Damiani, M.Dezani-Ciancaglini, P.Giannini. More dynamic object re-classification: $Fickle_{II}$, *ACM Transactions On Programming Languages and Systems* **24**(2), 2002, 153–191.

[HHJT98] U.Hensel, M.Huisman, B.Jacobs, H.Tews. Reasoning about Classes in Object-Oriented Languages: Logical Models and Tools, *European Symposium on Programming*, C.Hankin ed., Springer LNCS **1381**, 1998, 105–121.

[HL95] F.Honsell, M.Lenisa. Final Semantics for Untyped Lambda Calculus, *TLCA'95* Conf. Proc., M.Dezani, G.Plotkin eds., Springer LNCS **902**, Berlin 1995, 249–265.

[HLMP98] F.Honsell, M.Lenisa, U.Montanari, M.Pistore. Final Semantics for the π-calculus, *PROCOMET'98*, D. Gries et al. eds, Chapman & Hall, 1998.

[Jac96] B.Jacobs. Objects and Classes, co-algebraically, *Object-Orientation with Parallelism and Book Persistence*, B.Freitag et al. eds., Kluwer Academic Publishers, 1996, 83–103.

[Jac97] B.Jacobs. Behaviour-refinement of object-oriented specifications with coinductive correctness proofs, *TAPSOFT'97*, M.Bidoit, et. al. eds., Springer LNCS **1214**, 1997, 787–802.

[Jac97a] B.Jacobs. Invariants, Bisimulations and the Correctness of Coalgebraic Refinements, *Algebraic Methodology and Software Technology*, M.Johnson ed., Springer LNCS **1349**, 1997, 276–291.

[JR96] B.Jacobs, J.Rutten. A tutorial on (co)algebras and (co)induction, *Bulletin of the EATCS* **62**, 1996, 222–259.

[Len96] M.Lenisa. Final Semantics for a Higher Order Concurrent Language, *CAAP'96*, H.Kirchner et. al. eds., LNCS **1059**, 1996, 102–118.

[Rei95] H.Reichel. An approach to object semantics based on terminal co-algebras, *MSCS* **5**, 1995, 129-152.

[Tew00] H.Tews. Coalgebras for Binary Methods, *CMCS'2000*, ENTCS **33**, 2000.

[Rut00] J.J.M.M.Rutten. Universal coalgebra: a theory of systems, *TCS* **249**(1), 2000, 3–80.

[RT94] J.J.M.M.Rutten, D.Turi. *REX* Conference Proceedings, J.de Bakker et al. eds., LNCS **803**, 1994, 530–582.

[ST97] D.Sannella, A.Tarlecki. Essential concepts of algebraic specification and program development, *Formal Aspects of Computing* **9**, 1997, 229–269.

[TP97] D.Turi, G.Plotkin. Towards a mathematical operational semantics, 12^{th} *LICS*, IEEE, Computer Science Press, 1997, 280–291.

A Coalgebraic Preliminaries

In this section, we recall the notion of *coalgebra*, *coalgebra morphism*, *coalgebraic bisimulation*, and the main result of the coalgebraic paradigm, which characterizes equivalences induced by morphisms into final coalgebras as *coalgebraic bisimilarities*, i.e. *greatest* coalgebraic bisimulations. For more details, see e.g. [JR96].

Definition 7. *Let $F : \mathcal{C} \to \mathcal{C}$. A* F-coalgebra *is a pair (X, α_X), where $\alpha_X : X \to FX$ is an arrow in $\mathcal{C}$. F-coalgebras can be endowed with the structure of a category by defining F-coalgebra morphisms as follows. $f : (X, \alpha_X) \to (Y, \alpha_Y)$ is an* F-coalgebra morphism *if $f : X \to Y$ is an arrow of the category $\mathcal{C}$ such that the following diagram commutes*

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ F(X) & \xrightarrow[F(f)]{} & F(Y) \end{array}$$

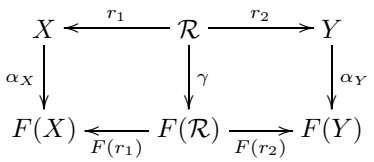
Before introducing the notion of *F-bisimulation*, we introduce the notion of span:

Definition 8. *A* span *$(\mathcal{R}, r_1, r_2)$ on objects X, Y consists of an object $\mathcal{R}$ in $\mathcal{C}$, and two ordered arrows, $r_1 : \mathcal{R} \to X$ and $r_2 : \mathcal{R} \to Y$.*
Spans on objects X and Y can be ordered as follows:

$$(\mathcal{R}, r_1, r_2) \leq (\mathcal{R}', r_1', r_2') \iff \exists f : \mathcal{R} \to \mathcal{R}'.\ \forall i = 1, 2.\ r_i = r_i' \circ f\ .$$

The notion of binary relation is expressed, in a general categorical setting, as an equivalence class of monic spans. As pointed out in [TP97], F-bisimulations on F-coalgebras can be simply taken to be spans in the category of F-coalgebras. The following definition due to [TP97], generalizes the original definition of [AM89].

Definition 9 (F-bisimulation). *Let F be an endofunctor on the category $\mathcal{C}$. A span $(\mathcal{R}, r_1, r_2)$ on objects X, Y is an* F-bisimulation *on the F-coalgebras (X, α_X) and (Y, α_Y), if there exists an arrow of $\mathcal{C}$, $\gamma : \mathcal{R} \to F(\mathcal{R})$, such that $((\mathcal{R}, \gamma), r_1, r_2)$ is a coalgebra span, i.e.*



In *Set* one often only considers bisimulations which are relations, i.e. spans $(\mathcal{R}, r_1, r_2)$ for a relation $\mathcal{R} \subseteq X \times Y$. Notice that every span $(\mathcal{R}, r_1, r_2)$ in *Set* can be regarded as representing the image $\langle r_1, r_2 \rangle(\mathcal{R}) \subseteq X \times Y$. The order on spans corresponds to relational inclusion of images. Furthermore, the image of a (span) bisimulation is a (relational) bisimulation (see e.g. [Rut00], Lemma 5.3).

When the two F-coalgebras (X, α_X) and (Y, α_Y) in the definition above coincide, we will simply say that the span is an F-bisimulation on the F-coalgebra (X, α_X).

The following theorem generalizes the fact that, in set-theoretic categories, equivalences induced by unique morphisms into final coalgebras can be characterized coinductively as the *greatest* F-bisimulations.

Theorem 4. *Suppose that $F : \mathcal{C} \to \mathcal{C}$ has a final F-coalgebra $(\Omega_F, \alpha_{\Omega_F})$. Let (X, α_X) be a F-coalgebra, and let $\mathcal{M} : (X, \alpha_X) \to (\Omega_F, \alpha_{\Omega_F})$ be the unique final morphism. If F preserves weak pullbacks, then*
i) for all F-bisimulations $(\mathcal{R}, r_1, r_2)$ on (X, α_X), $\mathcal{M} \circ r_1 = \mathcal{M} \circ r_2$;
ii) the kernel pair of $\mathcal{M}$ is an F-bisimulation on (X, α_X).