Coalgebraic Semantics and Observational Equivalences of an Imperative Class-based OO-Language[⋆]

Furio Honsell[1] Marina Lenisa[2,3]

Dipartimento di Matematica e Informatica, Università di Udine,
Via delle Scienze 206, 33100 Udine, ITALY.

Rekha Redamalla[4]

Dipartimento di Matematica e Informatica, Università di Udine,
Via delle Scienze 206, 33100 Udine, ITALY,
and B.M. Birla Science Center,
Adarsh Nagar, Hyderabad, 500 063 A.P., INDIA.

Abstract

Fickle is a class-based object oriented imperative language, which extends Java with object *re-classification.* In this paper, we introduce a natural *observational equivalence* on *Fickle* programs. This is a *contextual* equivalence on *main* methods with respect to a given sequence of class definitions, i.e. a program. To study it, we use the formal computational model for OO-programming based on *coalgebras*, which has recently emerged, whereby objects are taken to be equal when the actions of methods on them yield the same *observations* and equivalent next states. However, in order to deal with *imperative features*, we need to extend the original approach of H.Reichel and B.Jacobs in various ways. In particular, we introduce a *coalgebraic* description of objects (states of a class), which induces a *coinductive behavioural equivalence* on programs. For simplicity, we focus on Fickle *objects* whose methods do not take more than one object parameter as argument. Completeness results as well as problematic issues arising from *binary methods* are also discussed.

Key words: Imperative class-based OO-programming, observational equivalences, coalgebraic semantics, coinductive behavioural equivalences.

Introduction

In the global computing community, there has been growing interest in *class-based object oriented* languages. Despite this, however, relatively little work has been done on program equivalence for such languages. This is due probably also to the fact that no formal model for OO-programming has been generally accepted. In recent years, Reichel and Jacobs [13,9] have introduced such a model based on coalgebras. The idea underpinning this approach is that *coalgebras*, duals of algebras, allow to focus on the behaviour of objects, while abstracting from the concrete representation of Self. This approach has been used mainly for extending logical specification, and program and data refinement techinques to OO-languages (see [10,5]).

This paper intends to initiate an investigation programme into the possibility of utilizing the coalgebraic computational model also for program equivalence and program transformation. This is a somewhat dual goal w.r.t. the program refinement of Reichel and Jacobs. To this end, we need to address directly critical issues pertaining to *imperative* features, as well as *binary methods*, i.e. methods taking more than one object argument as parameter.

We focus on a fragment of the imperative typed class-based language *Fickle*, [4], which extends *Java* with *re-classification*. Re-classification allows objects to change class membership dynamically, while retaining their identity. We consider Fickle as a representative of this class of typed imperative class-based object oriented languages, and the results in this paper apply also to Java and similar languages.

We study the (family of) contextual equivalances $\approx_P$ (indexed by program P) on Fickle expressions (i.e, bodies of *main* methods). To this aim, we utilize a coalgebraic *behavioural equivalence* on Fickle *objects*. Our main result is an *adequate coalgebraic semantics* of Fickle expressions w.r.t. equivalences $\approx_P$.

Coalgebraic semantics originated with Aczel-Mendler, Rutten-Turi, for CCS-like languages, [1,2,15], and it was further generalized to λ-calculus, [6], higher-order imperative languages, [12], object-oriented languages in a functional setting, [13,9], π-calculus, [7].

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

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

[1] Email:honsell@dimi.uniud.it.

[2] Email:lenisa@dimi.uniud.it.

[3] corresponding author.

[4] Email:rrekhareddy@yahoo.com.

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. For preliminaries on coalgebraic semantics, we refer to [11].

However, in the original coalgebraic approach only a single class *in isolation* is considered and the setting is purely *functional*.

In dealing with Fickle, the approach of [13,9] needs to be refined to accomodate *imperative* features as well as general *programs*, i.e. sequences of classes possibly related by inheritance, mutual definitions, etc. Special care needs to be devoted to representing the *store*, and in defining the evolution of objects, we have to take into account all possible pointers involving them.

For the sake of simplicity, we deal first only with *non-binary* methods, i.e. methods which take no more than one class argument. Binary methods cannot be treated simplistically, since they produce *contravariant* occurrences of the variable in the corresponding functor. Extensions of the coalgebraic paradigm to mixed functors have been considered in [14], but such extensions are rather complex and cover only a restricted range of cases. We briefly sketch an alternative approach to dealing with binary methods, based on representing functions as *graphs*. This approach is completely satisfactory in a purely functional setting. However, in an imperative setting, binary methods bring about further problematic issues which we briefly touch upon.

Interestingly, the coalgebraic equivalence on Fickle objects induces a behavioural equivalence on Fickle expressions, which can be used to study notions of *observational equivalences*. In this paper we use the coalgebraic equivalence to study the *contextual equivalence* $\approx_P$, that we introduce. This is an equivalence on Fickle *expressions*, viewed as the bodies of the *main* method, with respect to a given program P, i.e. a collection of class definitions. Two *main* methods are equated if and only if they have the same behaviour in any context, w.r.t. the given program P.

Synopsis. In Section 1, we recall the syntax and the operational semantics of Fickle and we introduce the observational equivalence on expressions (programs). In Section 2, we give the coalgebraic description of Fickle programs, together with the induced behavioural equivalence. In Subsection 2.1, we discuss binary methods. In Section 3, we compare the coalgebraic equivalence with the observational equivalence on expressions introduced before. Final remarks and directions for future work appear in Section 4.

Acknowledgements.

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 [4] 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 (possibly *abstract*) 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 class. Root classes abstract over state classes. While (state) classes consist of a sequence of fields and methods, in abstract (root) classes, some methods can only be declared. 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. Hence, fields cannot be reclassified. In contrast, the type of *this* and parameters may be a state or root class, i.e. these variables may be reclassified. However, although these restrictions are important in order to define a sound type system (see [4]), they are not strictly necessary to our purposes.

Objects are created with the expression **new**c, where c is any class. Re-classification expressions, $id \Downarrow c$, set the class of id to c, where c must be a state class.

Methods declarations have the shape:

$$t\ m\ (t_1 x_1, \ldots, t_q x_q)\{c_1, \ldots, c_n\}\{\ e\ \}$$

where t is the result type, $t_1, \ldots, t_q$ are the types of the formal parameters $x_1, \ldots, x_q$ and e is the body. The list of root classes $c_1, \ldots, c_n$ 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*. Moreover, we assume no local variables in method bodies.

In Table 1, summarizing Fickle syntax, we have omitted the syntax of boolean and integer expressions, which involves the standard operators.

Finally, we assume the inheritance hierarchy to be a tree, root classes to extend only non-root and non-state classes, and state classes to extend either root classes or state classes. For examples of Fickle programs, we refer to [3,4].

progr	:=	class*
class	:=	[**state**] **class** c **extends** c { field* meth* }
absclass	:=	[**root**] **class** c **extends** c { field* meth* mdecl* }
field	:=	type f
meth	:=	type m (par*) eff { e }
mdecl	:=	type m (par*) eff
type	:=	**bool** | **int** | c
par	:=	type x
eff	:=	{ c* }
expr (∋) e	:=	**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** | 0 | 1 . . .
id	:=	**this** | x

with the following conventions

c	::=	c | c' | c_i | d | . . .	for class names
f	::=	f | f_i | . . .	for field names
m	::=	m | m_i | m_{ij} | . . .	for method names
x	::=	x | y | z | . . .	for parameter names

Table 1
Syntax of Fickle

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 type of the rewriting relation is:

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

where:

$$\begin{array}{lcl}
addr & \triangleq & Nat \\
val & \triangleq & sVal \cup addr \\
dev & \triangleq & \{ \ nullPntrExc, \ \ stuckErr \ \} \\
object_c & \triangleq & \{[f_1 : v_1, \ldots, f_r : v_r]^c \mid \ f_1, \ldots, f_r \in fid_c \text{ are} \\
 & & \ \text{the field identifiers of c, } v_1, \ldots, v_r \in val \ \} \\
object & \triangleq & \bigcup_c object_c \\
store & \triangleq & (\{this\} \rightarrow addr) \times (varid \rightarrow_{pfin} val) \times (addr \rightarrow_{pfin} object) \ ,
\end{array}$$

where $sVal$ is defined in Table 1, *varid* is the set of variable identifiers, fid_c is the set of field identifiers of c, and $\rightarrow_{pfin}$ denotes the space of partial functions with finite domain. Notice that an element of $object_c$ is in fact a partial function in $(fid_c \rightarrow_{pfin} val)$.

In particular, stores are partial functions with *finite* domain, mapping *this* to an address, variables of base type to values, variables of class type to addresses, and addresses to objects. 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 addresses with ι, stores with σ, values with v, objects with o, exceptions and errors with dv.

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 v, address ι, identifier or address z, field identifier 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*: $o[v/f] \triangleq [f_1 : v_1, \ldots, f_l : v, \ldots f_r : v_r]^c$, where $f_l = f$ for some $l \in 1, \ldots, r$,
- *store update*: $\sigma[v/z](z) = v, \quad \sigma[v/z](z') = \sigma(z')$ if $z' \neq z$.

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

Tables 2 and 3 list the rewriting rules of the operational semantics.

For lack of space, in Table 3 some rules are compressed in a single one (see e.g. the first rule). Notice that the rules as presented in Table 3 are non-deterministic, but common sense suggests how to resolve ambiguities.

The evaluation of the expression **new** c in a store σ extends σ with a new *canonical address*. Moreover, all fields of the new object are initialized with canonical values, which we assume, by convention, to be **false** and 0 for boolean and integer fields, respectively, and **null** for fields of class type. The function $\mathcal{F}_S$ used in the rule for **new** (and for re-classification) is such that $\mathcal{F}_S(P, c)$ returns the set of fields defined in the class c, while $\mathcal{F}_S(P, c, f)$, used in the rule for reclassification, gives the type of the field f in class c.

In the rule for method call, $e_0.m(e_1, \ldots, e_n)$ in Table 2, we use the function $\mathcal{M}$: $\mathcal{M}(P, c, m)$ returns the definition of method m in class c going through the class hierarchy (see [4] for more details). Moreover, the premise $\sigma_n(\iota) = [...]^c$ means that, in the store σ'_n, the address ι refers to an object of the class c.

For re-classification expressions, $id \Downarrow d$, 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 (as in the case of **new** expressions). The term $\mathcal{R}(P, t)$, defined by

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

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.

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

$$\frac{\begin{array}{l}\sigma(x) \neq \mathit{Udf} \\ (e,\sigma) \longrightarrow_P (v,\sigma')\end{array}}{(x := e,\sigma) \longrightarrow_P (v,\sigma'[v/x])} \qquad \frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\iota,\sigma'') \\ (e',\sigma'') \longrightarrow_P (v,\sigma''') \\ \sigma'''(\iota)(f) \neq \mathit{Udf} \\ \sigma' \triangleq \sigma'''[\sigma'''(\iota)[v/f]/\iota]\end{array}}{(e.f := e',\sigma) \longrightarrow_P (v,\sigma')}$$

$$\frac{\begin{array}{l}(e_1,\sigma) \longrightarrow_P (v',\sigma'') \\ (e_2,\sigma'') \longrightarrow_P (v,\sigma')\end{array}}{(e_1;e_2,\sigma) \longrightarrow_P (v,\sigma')} \qquad \frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\iota,\sigma') \\ \sigma'(\iota)(f) \neq \mathit{Udf}\end{array}}{(e.f,\sigma) \longrightarrow_P (\sigma'(\iota)(f),\sigma')}$$

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

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

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0) \\ (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,n\}) \\ \sigma_n(\iota) = [\ldots]^c \\ \mathcal{M}(P,c,m) = t\ m(t_1 x_1,\ldots,t_n x_n)\ \phi\ \{\ e\ \} \\ \sigma' = \sigma_n[\iota/\text{this}, v_1/x_1,\ldots,v_n/x_n] \\ (e,\sigma') \longrightarrow_P (v,\sigma'')\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (v,\sigma''[\text{this} \mapsto \sigma_n(\text{this}), x_1 \mapsto \sigma_n(x_1),\ldots,x_n \mapsto \sigma_n(x_n)])}$$

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

Table 2
Operational Semantics: execution without exceptions and errors

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 finitely many holes. As observable values, we take values of base types and errors/exceptions, i.e. $\mathit{obsval} \triangleq \mathit{sVal} \cup \mathit{dev}$. With $(e,\sigma) \Downarrow_P u$

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

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

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

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

$$\frac{\begin{array}{l}(e_0,\sigma)\ \longrightarrow_P\ (v,\sigma_0)\\ v\neq\text{null}\\ v\notin\text{addr or }\sigma_0(v)=\mathit{Udf}\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma)\ \longrightarrow_P\ (\text{stuckErr},\sigma_0)}\qquad \frac{\begin{array}{l}(e,\sigma)\ \longrightarrow_P\ (\iota,\sigma'')\\ (e',\sigma'')\ \longrightarrow_P\ (v,\sigma')\\ \sigma'(\iota)(f)=\mathit{Udf}\end{array}}{(e.f:=e',\sigma)\longrightarrow_P\ (\text{stuckErr},\sigma')}$$

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

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

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

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

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

Table 3
Operational semantics: generation and propagation of exceptions and errors

we abbreviate the fact that there exists σ' such that $(e, \sigma) \rightarrow_P (u, \sigma')$, for $u \in sVal \cup dev$.

Definition 1.1 `(Contextual Equivalence)`:
Let $\approx_P \subseteq expr \times expr$ be defined by: $e \approx_P e' \overset{\Delta}{\Longleftrightarrow}$

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

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'. Notice that, by the assumption that all fields in a class are private (see Section 1.1), *main* methods can only access objects through class methods. In particular, in Definition 1.1 above, field access expressions appear neither in the expressions e, e' nor in the context $C[\,]$.

In the definition of the observational equivalence $\approx_P$ above, the program P is fixed. However, in many cases, e.g. in program refinement, we are interested in 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 only a sequence of method declarations. Then a program P_1 implements a program specification P, when the method declarations in each class of 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. By way of example, we introduce the following simple equivalence.

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 1.2 Let P_1, P_2 implement the same program specification P. We define the equivalence $\simeq$ by:

$$P_1 \simeq P_2 \ \overset{\Delta}{\Longleftrightarrow}\ \forall e\ \forall u \in obsval.\ (e, \emptyset) \Downarrow_{P_1} u \ \Leftrightarrow\ (e, \emptyset) \Downarrow_{P_2} u\ .$$

For lack of space, in this paper we will focus only on the contextual equivalence $\approx_P$ and we will not study the program equivalence of Definition 1.2. We just point out that our coalgebraic description of Fickle objects given in the next section induces also a notion of coalgebraic program equivalence, which would be interesting to compare with the program equivalence $\simeq$.

2 Coalgebraic Description of Fickle Objects and Programs

In this section, we give a coalgebraic account of Fickle objects (and programs) for the fragment of Fickle consisting of *unary methods*, i.e. methods which do not take more than one object parameter. Following [13,9], we model classes as coalgebras, where the carrier represents the objects 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.

In order to model the evolution of objects in an imperative setting, we need to account also for sharing of addresses in the store and aliasing of variables.

Our coalgebraic model naturally induces a *coinductive* equivalence on objects of a program P, which we will use in Section 3 to study the contextual equivalence introduced in Definition 1.1.

Finally, we briefly discuss the general case of binary methods. These are problematic to deal with, since they produce *contravariant* occurrences of the parameter in the functor modeling the program. We propose two possible approaches, in order to turn contravariant occurrences into covariant ones. The first consists in "freezing" the contravariant occurrence of X to the carrier of the coalgebra. The second consists in representing binary methods as *graphs* instead of functions. Under suitable conditions the two approaches coincide.

Moreover, in order to account for the behaviour of objects under binary methods, we need to extend the original approach, by describing not only the behaviour of the objects under application of methods of their class, but also under application of any other binary method in the program which uses such objects as parameters.

Finally, in dealing with binary methods in an imperative setting we need to take care of possible inconsistencies of referenced values by object parameters. We start by defining our representation of imperative objects of a class c.

Definition 2.1 Let $\mathit{refobject}_c$ be the set of pairs (ι, O), where $\iota \in \mathit{addr}$, and $O \in (\mathit{addr} \to_{\mathit{pfin}} \mathit{object})$ is the least *closed* function, — i.e. $\forall o \in \mathrm{range}(O).\ \forall \iota \in \mathrm{range}(o).\ \iota \in \mathrm{dom}(O)$ —, such that $\iota \in \mathrm{dom}(O)$ and $O(\iota) \in \mathit{object}_c$.

Essentially, a refobject can be viewed as a *minimal* store induced by an address, when we do not consider the environment part.

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

Before introducing our coalgebraic semantics, we need to define the notion of *consistency* between refobjects and stores, the notion of store update with a refobject, and the notion of refobject induced by an address in a store:

Definition 2.2 Let $\sigma \in \mathit{store}$, $O \in \mathit{refobject}_c$, $\iota \in \mathit{addr}$.
i) O and σ are *consistent*, written $\mathit{con}(O, \sigma)$, if for all addresses $\iota \in \mathit{dom}(O)$, if $\iota \in \mathit{dom}(\sigma)$, then $O(\iota) = \sigma(\iota)$.
ii) For O and σ consistent, and $x \in \mathit{id}$, we define $\sigma[O/x]$ the store σ in which the object corresponding to the *refobject* O has been associated to x, and the rest of the store, if necessary, has been updated according to O.
iii) Let $\iota \in \mathit{dom}(\sigma)$. We denote by $\overline{\sigma(\iota)}$ the unique refobject (ι, O) included in

σ. Let $x \in \mathit{varid} \cup \{\mathit{this}\}$. We denote by $\overline{\sigma(x)}$, $\sigma(x)$ itself, if x has base type, the unique refobject $(\sigma(x), O)$ included in σ, otherwise.

Now we introduce the coalgebraic description of the fragment of Fickle consisting of unary methods. To this aim, we endow the set of refobjects of a given program P with a coalgebra structure for the functor induced by the methods in P. A method $t_0\ m(t_1 x_1, \ldots, t_q x_q)$ in P, when called on an object together with a list of actual parameters, can either terminate (successfully or with an exception/error) producing a possibly modified object, or not terminate. The behaviour of methods on objects determines the coalgebraic structure:

Definition 2.3 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 $F : \mathit{Set} \to \mathit{Set}$ be defined by

$$F \triangleq \coprod_i \prod_j F_{ij} \ ,$$

where $F_{ij} : \mathit{Set} \to \mathit{Set}$ is determined by the method declaration

$$t_0\ m_{ij}\ (t_1 x_1, \ldots, t_q x_q)\ \{c'_1, \ldots, c'_p\}$$

of the class c_i as follows:

$$F_{ij} X \triangleq [\![t_1]\!] \times \ldots \times [\![t_q]\!] \to (([\![t_0]\!] + \mathit{dev}) \times X + 1) \ ,$$

where, for all $i = 0, \ldots, q$,

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

The definition of F_{ij} on arrows is canonical.
ii) Let us denote $\coprod_{c_i} \mathit{refobject}_{c_i}$ simply by $\mathit{refobject}_P$. Let $\alpha_P : \mathit{refobject}_P \to F(\mathit{refobject}_P)$ be defined by

$$\alpha_P \triangleq [\langle \alpha_{ij} \rangle_j]_i \ ,$$

where $\alpha_{ij} : \mathit{refobject}_{c_i} \to F_{ij}(\coprod_{c'_k \in C_{ij}} \mathit{refobject}_{c'_k})$, for C_{ij} the set of classes to which the method m_{ij} can reclassify the object, is defined by:

$$\alpha_{ij}(O) \triangleq \boldsymbol{a} \mapsto \begin{cases} (u, \overline{\sigma_1(\mathit{this})}) & \text{if } e \text{ is the body of } m_{ij} \text{ and} \\ & (e, \emptyset[O/\mathit{this}, \boldsymbol{a}/\boldsymbol{x}]) \longrightarrow_P (u, \sigma_1) \\ * & \text{otherwise} \ , \end{cases}$$

where $*$ denotes the only element of 1. Notice that the store $\emptyset[O/\mathit{this}, \boldsymbol{a}/\boldsymbol{x}]$ is always defined (i.e. there are no consistency problems), since all actual parameters are of base type.

iii) Let $[\![\ ]\!]^F_P : (\mathit{refobject}_P, \alpha_P) \to (\Omega_F, \alpha_{\Omega_F})$ be the coalgebraic semantics, i.e. the unique F-coalgebra morphism into the *final* F-coalgebra.

By applying the general theory of coalgebraic semantics, we get the following coinductive characterization of the equivalence induced by $[\![\ ]\!]^F_P$:

Proposition 2.4 *The coalgebraic semantics $[\![\ ]\!]^F_P$ induces the following behavioural equivalence on objects of P: for all $O, O' \in \mathit{refobject}_c$, where c is a class of P,*

$$O {\sim}^F_P O' \iff$$

$\forall$ method $m(\boldsymbol{x})$ in c with body e, $\forall$ list of arguments $\boldsymbol{a}$ for $\boldsymbol{x}$,
$\underline{(e, \emptyset[O/\mathit{this}, \boldsymbol{a}/\boldsymbol{x}])} \longrightarrow_P (u, \sigma_1) \;\Rightarrow\; (e, \emptyset[O'/\mathit{this}, \boldsymbol{a}/\boldsymbol{x}]) \longrightarrow_P (u, \sigma'_1) \;\wedge\; \sigma_1(\mathit{this}) {\sim}^F_P \sigma'_1(\mathit{this})$*, and conversely.*

Corollary 2.5 *${\sim}^F_P$ is the greatest fixed point of the following monotone (w.r.t. subset inclusion) operator on relations on refobjects:*

$\Phi(R) \triangleq \{(O, O') \mid \forall m(\boldsymbol{x}) : e$ *in* c, $\forall$ *list of arguments* $\boldsymbol{a}$ *for* $\boldsymbol{x}$,
$\underline{(e, \emptyset[O/\mathit{this}, \boldsymbol{a}/\boldsymbol{x}])} \longrightarrow_P (u, \sigma_1) \;\Rightarrow\; (e, \emptyset[O'/\mathit{this}, \boldsymbol{a}/\boldsymbol{x}]) \longrightarrow_P (u, \sigma'_1) \;\wedge\; \sigma_1(\mathit{this})\ R\ \sigma'_1(\mathit{this})$, *and conversely* $\}$.

In other words, the following coinduction principle for establishing ${\sim}^F_P$ is sound and complete:

$$\frac{ORO' \;\wedge\; R \textit{ is a } \Phi\textit{-bisimulation}}{O \sim^F_P O'} \quad ,$$

where a Φ-bisimulation R is a relation s.t. $R \subseteq \Phi(R)$.

Example 2.6 i) Let *Register* be a class with just one field containing the integer value of a register, and two methods, *getval* and *setval*. The first method returns the contents of the register, the latter sets the contents to a new value passed as parameter, and returns the new value. One can easily check that the coalgebraic equivalence on objects class *Register* equates two registers if and only if they have the same contents.
ii) Let *IntList* be a class representing possibly circular lists of integers. The class *IntList* has two fields, representing the head and the tail of a list, i.e. containing an integer value and a list, respectively, and two methods, returning the head and the tail of a list. Then the coalgebraic equivalence on *IntList* equates two lists if and only if they have the same value in the head and the same address in the tail.
iii) In order to recover the extensional equivalence on lists, one can define the class *IntList* by considering just one method, taking an integer n as parameter and returning the value of the n-th element of a list.

The coalgebraic equivalence ${\sim}^F_P$ equates objects which behave in the same way under method application, for all lists of parameters, in the *minimal* store. Actually, store minimality is not relevant. Namely, one can easily show that the behaviour of an object only depends on method parameters, and not on the rest of the store, if we assume that the expression **new** c does not appear in the bodies of class methods. However, we conjecture that the above assumption can be eliminated. Anyway, we feel that this is not a strong assumption, since usually class methods are used to access or modify objects, while creation of new objects is performed in the *main* method.

Moreover, in what follows, we tacitly assume also that, if two objects of a root class d are $\sim_P^F$-equivalent, then their canonical extensions (via re-classification) to objects of a state subclass c are still $\sim_P^F$-equivalent. This means that in the subclass c there are no extra methods which discriminate solely on the basis of the fields in the superclass d. This is quite a natural hypothesis, which is necessary to deal with re-classification.

Thus we have:

Lemma 2.7

$$O \sim_P^F O' \iff$$

$\forall$ *method* $m(\boldsymbol{x})$ *in* c *with body* e, $\forall\sigma.\ con(O,\sigma)\ \wedge\ con(O',\sigma)$, $(e,\sigma[O/this]) \longrightarrow_P (u,\sigma_1)\ \Rightarrow\ (e,\sigma[O'/this]) \longrightarrow_P (u,\sigma'_1) \wedge \overline{\sigma_1(this)} \sim_P^F \overline{\sigma'_1(this)}$, *and conversely.*

The equivalence $\sim_P^F$ on refobjects naturally induces an equivalence on stores, if we take stores to be equivalent on all variables:

Definition 2.8 We define

$$\sigma \sim_P^F \sigma' \ \stackrel{\Delta}{\iff}\ \forall x \in id.\ \overline{\sigma(x)} \sim_P^F \overline{\sigma'(x)}\ .$$

Another immediate consequence of the fact that object behaviour only depends on method parameters, is that, if two objects are $\sim_P^F$-equivalent, then they behave in the same way under application of methods on $\sim_P^F$-equivalent parameters:

Lemma 2.9

$$O \sim_P^F O' \iff$$

$\forall\, m(\boldsymbol{x}) : e$ *in* c, $\forall\sigma,\sigma'.\ \sigma \sim_P^F \sigma'\ \wedge\ con(O,\sigma) \wedge con(O',\sigma'),\ (e,\sigma[O/this]) \longrightarrow_P (u,\sigma_1)\ \Rightarrow\ (e,\sigma'[O'/this]) \longrightarrow_P (u,\sigma'_1)\ \wedge\ \overline{\sigma_1(this)} \sim_P^F \overline{\sigma'_1(this)}$, *and conversely.*

2.1 Binary Methods

If a method m_{ij} in Definition 2.3 is binary, then clearly the functor F_{ij} (and hence F) is not covariant. An example of a binary method is the method equal : $c \times c \to bool$, which takes another instance of the class as argument. The second occurrence of c produces a contravariant occurrence of X in $FX \triangleq \ldots \times (\mathbf{X} \to ((bool + dev) \times X) + 1) \times \ldots$. Therefore, the coalgebraic approach does not apply directly in this case.

A first solution for turning contravariant occurrences in the functor F in covariant ones consists in "freezing" the contravariant occurrence of X to the carrier $refobject_P$ of the coalgebra. For example, in the case of the method *equal*, we could take $GX \triangleq \ldots \times (refobject_P \to ((bool + dev) \times X) + 1) \times \ldots$.

An alternative approach consists in modeling binary methods as *graphs* instead of functions, by substituting the powerset functor in place of the function space. In the case of the method *equal* this would correspond to take $HX \triangleq \ldots \times \mathcal{P}(X \times ((\mathit{bool} + \mathit{dev}) \times X) + 1) \times \ldots$.

Moreover, in both cases, in order to describe correctly the behaviour of objects under application of binary methods, we need to extend the original coalgebraic approach and consider in the definition of the functor also (binary) methods, of possibly different classes, to which the object is supplied as a parameter. Otherwise we loose the analogue of Lemma 2.9. Namely, let us consider the following trivial counterexample. Let A be a class with only one field of type *bool*, and just one (binary) method m taking as argument another object of the class A, and returning the value of the field of this argument. Then all objects of A are coalgebraic equivalent, if we do not consider the behaviour of objects as (ordinary) arguments of m. However, the method m applied to the same object together with arguments with different values in the field gives non-equivalent results.

In general, the equivalence induced by G is included in the equivalence induced by H, but not vice versa. In order to make the two approaches coincide with the "intended contravariant" equivalence, one has to compensate the "observability deficit" of unary methods. Roughly, one has to add in the class a sufficient number of unary methods allowing to observe all the fields used by binary methods. As a consequence, the coalgebraic equivalence is determined solely by unary methods.

This programme works out smoothly in a functional setting, [8]. However, in our imperative setting, binary methods give rise to the extra issue of possible inconsistencies between the object O and the other object parameters, even in the empty store. In particular, if we consider the natural extension of the coalgebraic semantics of unary methods to binary methods, we get an object equivalence which discriminates on the basis of addresses, both in the case of the freezing functor G and in the case of the graph functor H. Namely, let us focus on freezing, and let us consider the class *Register* of Example 2.6, extended with the binary method *add*, which adds the contents of two registers. Then the method *add* tells apart registers with different addresses but equal contents, when we apply it to a register parameter consistent with e.g. the first register but not with the second one. To overcome this problem, one could modify the notion of object equivalence, by testing the behaviour of objects under method application only on parameters consistent with both objects. However, somewhat surprisingly, this is not a transitive relation, in general. A possible solution to the transitivity problem above consists again in compensating the observability deficit of unary methods. However, this deserves further study, and we leave it as an open problem how to give a coalgebraic description of Fickle objects in the general case.

3 Coalgebraic and Observational Equivalences on Programs

In this section, we introduce a notion of equivalence on expressions w.r.t. a program P, $\dot{\approx}_P$, which is induced by the coalgebraic equivalence on objects $\sim^F_P$ of Section 2, and we briefly discuss the relationships between $\dot{\approx}_P$ and the contextual equivalence $\approx_P$ of Section 1.3.

From now on we denote the equivalence $\sim^F_P$ simply by $\sim_P$. The coalgebraic equivalence on objects introduced in the previous section naturally induces a notion of equivalence on expressions representing bodies of *main* methods w.r.t. a given program P. Two *main* methods are equivalent w.r.t. P when, for any store, they produce equivalent values and equivalent stores:

Definition 3.1 Let $\dot{\approx}_P \subseteq \mathit{expr} \times \mathit{expr}$ be defined by:

$$e \mathrel{\dot{\approx}_P} e' \iff$$

$\forall \sigma.\ ((e, \sigma) \rightarrow_P (u, \sigma_1) \implies (e', \sigma) \rightarrow_P (u', \sigma'_1)\ \wedge\ \overline{u} \sim_P \overline{u}'\ \wedge\ \sigma_1 \sim_P \sigma'_1)$,
and conversely,
where $\overline{u}$ is u, if $u \in \mathit{sVal} \cup \mathit{dev}$, and it is $\overline{\sigma_1(u)}$, if $u \in \mathit{addr}$.

We conjecture that $\dot{\approx}_P$ is adequate w.r.t. the contextual equivalence $\approx_P$, i.e:

Conjecture 3.2 *(Adequacy of $\dot{\approx}_P$): $\dot{\approx}_P \subseteq \approx_P$.*

Completeness of $\dot{\approx}_P$ trivially fails, because in the contextual equivalence there is no way of observing different addresses generated by **new** expressions. For instance, if the class c in the program P is s.t. only objects with the same address are $\sim_P$-equivalent, then the expressions $e \triangleq x :=$ **new** c and $e' \triangleq x :=$ **new** $c; x :=$ **new** c are *not* $\dot{\sim}_P$-equivalent. However, there is no context separating them, since in the observational equivalence $\approx_P$ we only observe values of base types. Nevertheless, we can still get a completeness result for the restricted set of expressions not containing **new** expressions, under the assumption that in each class of the program there is an observable and modifiable field of base type:

Theorem 3.3 *(Completeness of $\approx_P$): Let P be a program, and let e, e' be* main *methods for P. If e, e' do not contain* ***new*** *expressions, and in each class c of P there is a field f of base type, a method m_1, which returns the value of f, and a method m_2, which sets the field f to a value given as parameter, then $e \approx_P e' \implies e \mathrel{\dot{\approx}_P} e'$.*

Proof. Assume by contradiction $e \approx_P e'$, but $e \mathrel{\dot{\not\approx}_P} e'$. The difficult case is when $e \mathrel{\dot{\not\approx}_P} e'$ because $\exists \sigma.\ (e, \sigma) \rightarrow_P (\iota, \underline{\sigma_1) \wedge} (e', \underline{\sigma) \rightarrow_P} (\iota', \sigma'_1)$, but $\overline{\iota} \not\sim_P \overline{\iota}'$ (if returned values are equivalent, but $\exists x.\ \overline{\sigma_1(x)} \not\sim_P \overline{\sigma'_1(x)}$, then we proceed as in the previous case by considering $C[\,] \triangleq [\,]; x)$. Let us assume that the objects to which ι, ι' point in the stores σ_1, σ'_1 are of class c. If $\sigma_1(\iota).f \neq \sigma'_1(\iota').f$, then the context $C[\,] \triangleq [\,].m_2(\ldots)$ tells apart e and e', getting a contradiction. Otherwise, if $\sigma_1(\iota).f = \sigma'_1(\iota').f$, then let z be fresh, and let us consider the store $\sigma[\iota/z]$. Then the context $C[\,] \triangleq z.m_2(\ldots a \ldots); ([\,].m_1(\ldots) = z.m_1(\ldots))$, where a is a new value for the field f, tells apart e and e' in the store $\sigma[\iota/z]$.$\square$

Notice that the assumption of no occurrence of **new** expressions in e, e' is fundamental in the proof above, since the technique of extending the store with a fresh variable would not work in the case the addresses ι, ι' are generated by **new** expressions.

4 Final Remarks and Directions for Future Work

In this paper we have introduced a contextual equivalence on Fickle programs, and we have defined a coalgebraic behavioural equivalence, which we conjecture to be adequate w.r.t. the contextual equivalence, and which is complete under suitable restrictions on syntax. To achieve this we had to extend the coalgebraic framework of [13,9] for OO-languages to the imperative case. Object re-classification is captured in a very natural way in the coalgebraic semantics.

In the future, we plan to:

- accomodate coalgebraically binary methods in the imperative setting;
- introduce coalgebraic equivalences between program implementations of the same specification, which approximate the program equivalence of Definition 1.2;
- explore the semantics of types in the coalgebraic setting;
- extend the coalgebraic description of Fickle programs of Section 2 to bialgebras, modeling *method constructors* as algebra operations;
- explore program logics, capitalizing on the coinduction principles supported by the coalgebraic semantics.

References

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

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

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

[4] 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.

[5] 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.

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

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

[8] F.Honsell, M.Lenisa, R.Redamalla. Coalgebraic Semantics of Binary Methods for a Functional OO-Language, in preparation.

[9] 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.

[10] 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.

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

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

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

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

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