$\mathcal{F}ickle$: Dynamic Object Re-classification ⋆ (Extended Abstract)

Sophia Drossopoulou[1], Ferruccio Damiani[2],
Mariangiola Dezani-Ciancaglini[2], and Paola Giannini[3]

[1] Department of Computing, Imperial College
[2] Dipartimento di Informatica, Università di Torino
[3] DISTA, Università del Piemonte Orientale

Abstract. *Re-classification* changes at run-time the class membership of an object while retaining its identity. We suggest language features for object re-classification, which could extend an imperative, typed, class-based, object-oriented language.
We present our proposal through the language $\mathcal{F}ickle$.[1] The imperative features combined with the requirement for a static and safe type system provided the main challenges. We develop a type and effect system for $\mathcal{F}ickle$ and prove its soundness with respect to the operational semantics. In particular, even though objects may be re-classified across classes with different members, they will never attempt to access non-existing members.

1 Introduction

In class-based, object-oriented programming, an object's behaviour is determined by its class. Case or conditional statements should be avoided when differences can be expressed through different classes. Thus, students paying reduced and employees paying full conference fees are best described through distinct classes Stdt and Empl with different methods fee().

However, this elegant approach does not scale when objects change classification. For example, how can we represent that mary, who was a Stdt, became an Empl? Usually, class based programming languages do not provide mechanisms for objects to change their class membership. Two solutions are possible: Either to replace the original Stdt object by an Empl object, or to merge the two classes Stdt and Empl into one, *e.g.* StdtOrEmpl.

Neither solution is satisfactory. The first solution needs to trace and inform all references to mary. The second solution blurs in the same class the differences in behaviour that were elegantly expressed through the class system. In fact, [20] lists the lack of re-classification primitives as the first practical limitation of object oriented programming.

We suggest language features which allow objects to change class membership dynamically, *e.g.* the object pointed at by variable aWindow belonged to class OpenedWindow but now belongs to class IconifiedWindow. We combine these features with a strong type system.

The idea of object re-classification is not new. From a foundational perspective, in [1] method overriding explains field update and delegation in an object-based calculus, while in [10] method extensions represent class inheritance. From a databases perspective, [2] suggests multiple most specific classes, while in [13] objects may accumulate several roles in a

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

[1] $\mathcal{F}ickle$ is the successor of an earlier proposal, $\mathcal{F}ickle$–99, [8]. Although both $\mathcal{F}ickle$ and $\mathcal{F}ickle$–99 address the same requirement for object re-classification, the approaches are very different.

functional setting. From a programming perspective, in [22] classes have "modes" representing different states, *e.g.* opened vs. iconified window. Wide classes [21] allow an object to be temporarily "widened" or "shrunk", *i.e.* become an object of a subclass or superclass, requiring run-time tests for the presence of fields. Predicate classes [5, 9] extend multimethods, suggesting method dispatch depending on predicates on the receiver and argument.

We take a programming perspective, and base our approach on an imperative, class-based language, where classes are types and subclasses are subtypes[2], and where methods are defined inside classes and selected depending on the class of the receiver. We achieve dynamic re-classification of objects by explicitly changing the class membership of objects.

We describe our approach through the language *Fickle*. A *re-classification* operation changes the class membership of an object while preserving its identity; it maintains all fields declared as common to the original and the target class and initializes the extra fields. *State* classes are possible targets of re-classifications; in that sense, they represent object's possible states. *Root* classes are the superclasses of such state classes and declare all the members common to them. Only non-state classes may appear as types of parameters or fields. *Fickle* is statically typed, with a type and effect system [17, 23], which determines the re-classification effect of an expression on the receiver and on all other objects. The type system is sound, so that terminating execution of a well-typed expression produces a value of the expected type, or a null-pointer exception, but does *not* get stuck.

This paper is organized as follows: In Section 2 we introduce *Fickle* informally using an example. In Section 3 we outline *Fickle*: the syntax, operational semantics, typing rules, and we state type soundness. In Section 4 we compare our proposal with other approaches. In Section 5 we describe design alternatives. In Section 6 we draw some conclusions.

A full version of this paper is available at http://www.di.unito.it/~damiani/papers/dor.html.

2 An example

In Figure 1 we define, using a Java-like syntax, a class Stack, with subclasses EmptyStack and NonEmptyStack. A stack has a capacity (field **int** capacity) that is the maximum number of integers it can contain and the usual functions characterizing stacks, *i.e.* isEmpty, pop, top, and push.

We have introduced two new kinds of classes: **state** and **root** classes. The state classes are the classes that may serve as targets of re-classifications. Such classes *cannot* be used as types for fields or parameters; in our example, EmptyStack and NonEmptyStack. The root classes define the fields and methods common to their state subclasses; in our example, Stack. The subclasses of root classes must be state classes. A state class c must have a (possibly indirect) root superclass c′; objects of class c may be re-classified to any subclass of c′.

Annotations like { } and {Stack} before throws clauses and method bodies are called *effects*. Effects list the root classes of all objects that may be re-classified by invocation of that method.

Methods with the empty effect { }, *e.g.* isEmpty, may not cause any re-classification. Methods with non-empty effects, *e.g.* pop and push with effect {Stack}, may re-classify objects of a subclass of their effect; in our case of Stack.

Such re-classifications may be caused by *re-classification expressions*, *e.g.* this⇓EmptyStack, or by further method calls.

[2] Even though the object-based paradigm may be more fundamental [1], and though classes should not be types, and subclasses should not imply subtypes [4], current praxis predominantly uses languages of the opposite philosophy.

```
class StackException extends Exception{
  StackException(String str) { } {super(str); }
}


abstract root class Stack extends Object{
  int capacity;    // maximum number of elements

  abstract bool isEmpty() {  } ;
  abstract int top() {  } throws StackException;
  abstract void push(int i) { Stack } throws StackException;
  abstract void pop()  { Stack } throws StackException;
}


state class EmptyStack extends Stack{
  EmptyStack(int n) {  } {capacity:= n;}

  bool isEmpty() {  } {return true;}
  int top() {  } throws StackException
    {throw new StackException("StackUnderflow");}
  void push(int i) { Stack }
    {this⇓NonEmptyStack; a:= new int[capacity]; t:= 0; a[0]:= i;}
  void pop() {  } throws StackException
    {throw new StackException("StackUnderflow");}
}


state class NonEmptyStack extends Stack{
  int[] a;    // array of elements
  int t;       // index of top element

  NonEmptyStack(int n, int i) {  }
    {capacity:= n; a:= new int[capacity]; t:= 0; a[0]:= i;}

  bool isEmpty() {  } {return false ;}
  int top() {  } {return a[t];}
  void push(int i) {  } throws StackException
    {t:= t + 1;
     if (t == capacity) throw new StackException("StackOverflow"); else a[t]:= i;  }
  void pop() { Stack }
    {if (t == 0) this⇓EmptyStack; else t:= t − 1;  }
}
```

Fig. 1. Program P_{st}- stacks with re-classifications

The method body of push in class EmptyStack contains the re-classification expression this⇓NonEmptyStack. At the start of the method the receiver is an object of class EmptyStack, therefore it contains the field capacity and does not contain the fields a and t. After execution of this⇓NonEmptyStack the receiver is of class NonEmptyStack, and therefore the fields a and t are accessible, while the field capacity retains its value. This mechanism allows the transmission of some information from the object before the re-classification to the object after the re-classification.

Consider example (1):

```
    Stack s;
1.  s:= new EmptyStack(100);
2.  s.push(3);
3.  s.push(5);
```
(1)

After line 1. the variable s is bound to an EmptyStack object, after line 2. the object (not the binding) is re-classified to NonEmptyStack. Therefore, the call of push in line 2. selects the method from EmptyStack, while the call of push in line 3. selects the method from NonEmptyStack. With the re-classification we allocate array a and initialize a and t.

Re-classification is transparent to aliasing. For instance, in example (2)

```
    Stack s1, s2;
1.  s1:= new NonEmptyStack(100, 3);
2.  s2:= s1;
3.  s1.pop();
4.  s2.isEmpty();
```
(2)

line 4. returns true. Re-classification removes from the object all fields that are not defined in its root superclass and adds the remaining fields of the target class. *e.g.* after line 3. in example (2) the object pointed at by s1 does not have the fields t and a.

Through aliasing, one re-classification might affect several variables; in example (2) the re-classification after line 3. affects both s1 and s2. For this reason, we prevent variables from accessing members declared in a state class, and we do that by forbidding state classes appearing in field or parameter declarations. Therefore, example (3) is illegal:

```
    Stack s;
    NonEmptyStack ns;      //  illegal!
1.  ns:= new NonEmptyStack(100, 3);
2.  s:= ns;
3.  s.pop();
4.  ns.t;                  //  error!
```
(3)

If the declaration of ns were legal, then, after line 3. the object bound to s and ns would be re-classified to EmptyStack, and the field access ns.t in line 4. would raise a fieldNotFound error. [3]

Therefore, members of state classes are only accessible via this either from methods of the particular state class if there is no previous mutation (*e.g.* access t in pop of class NonEmptyStack before the re-classification), or from methods of other state classes after appropriate re-classifications (*e.g.* access t in push of class NonEmptyStack after the re-classification).

State classes are used as types when typing the receiver, this. This allows accessing members declared in a state class, *e.g.* this.t in the body of push in class NonEmptyStack.

[3] An alternative, less satisfactory approach would forbid the assignment on line 2., *c.f.* Section 5.

3 The language *Fickle*

3.1 Syntax

A *Fickle* program is a sequence of class definitions, consisting of field and method definitions. Method bodies are sequences of expressions. We limit methods to have only one parameter called x. The syntax is similar to that of Java.

Class definitions may be preceded by the keyword **state**, or **root**. State classes describe the properties of an object while it satisfies some conditions; when it does not satisfy these conditions any more, it can be explicitly re-classified to another state class. For example, NonEmptyStack describes non-empty stacks; if these become empty, then they are re-classified to EmptyStack. Root classes abstract over state classes.[4] 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 class c′, where c′ must be a subclass of the uniquely defined root superclass of c. For example, Stack abstracts over EmptyStack and NonEmptyStack; objects of class EmptyStack may be re-classified to NonEmptyStack, and vice versa.

Objects of a non-state, non-root class c behave like regular Java objects, *i.e.* are never re-classified. However, objects pointed at by a variable x of type c *may* be re-classified. Namely, if c had two state subclasses d and d′ and x referred to an object of class d, the object may be re-classified to d′. Our type system insures that this re-classification will not cause accesses to fields or methods that are not defined for the object.

Fields, parameters and values returned by methods have declared types which are either boolean types or non-state classes; we call these types *variable types*. Thus, such fields and parameters may point to objects which do change class, but these changes do *not* affect their type. Instead, the type of `this` *may be a state class* and *may change*.

Objects are created with the expression **new** c − c may be *any* class, also a state class.

Re-classification expressions, this⇓c, set the class of `this` to c − c must be a state class.

Method declarations have the shape:

$$\mathsf{t\ m\ \ (t_1\ x)\ \{\ c_1, ..., c_n\ \}\ \{\ e\ \}}$$

where t is the result type, $\mathsf{t_1}$ is the type of the formal parameter x, and e the body. The effect consists of root classes $\mathsf{c_1}$,..., $\mathsf{c_n}$, with $\mathsf{n} \geq 0$.

We require root classes to extend only non-root and non-state classes, and state classes to extend either root classes or state classes. The judgment $\vdash\ \mathsf{P}\ \diamond_a$ expresses that program P satisfies these conditions, as well as the more obvious requirements for acyclic inheritance and unique definitions.

Remark 1. Section 2 allowed more liberal syntax: any number of parameters, abstract classes and methods, user defined constructors, local variables, exceptions, types **int** and **void**.

3.2 Operational semantics

We give a structural operational semantics that rewrites pairs of expressions and stores into pairs of values, exceptions, or errors, and stores - in the context of a given program. Stores map the unique parameter name x and the receiver `this` to values and addresses to objects. Values are booleans or addresses.

We discuss the two most significant rewrite rules of *Fickle*: method call and re-classification.

[4] Notice that root classes are not necessarily abstract classes and state classes may be superclasses. Thus, our proposal is orthogonal to the "abstract superclass rule" discussed in [14].

For method calls, $\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1)$, we evaluate the receiver e_0, obtaining an address, say ι. We then evaluate the argument, e_1. We find the appropriate body by looking up m in the class of the object at address ι – we use the function $\mathcal{M}(\mathsf{P},\mathsf{c},\mathsf{m})$ that returns the definition of method m in class c (going through the class hierarchy if needed). We then execute the body, after substituting `this` with the current object, and assigning to the formal parameter the value of the actual parameter. After the call, we restore the original receiver and parameter.[5]

$$\frac{\begin{array}{l}\mathsf{e}_0,\sigma \rightsquigarrow_{\mathsf{P}} \iota,\sigma_0\\ \mathsf{e}_1,\sigma_0 \rightsquigarrow_{\mathsf{P}} \mathsf{v}_1,\sigma_1\\ \sigma_1(\iota) = [[\ldots]]^{\mathsf{c}}\\ \mathcal{M}(\mathsf{P},\mathsf{c},\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x})\ \phi\ \{\ \mathsf{e}\ \}\\ \sigma' = \sigma_1[\mathtt{this}{\mapsto}\iota][\mathsf{x}{\mapsto}\mathsf{v}_1]\\ \mathsf{e},\sigma' \rightsquigarrow_{\mathsf{P}} \mathsf{v},\sigma''\end{array}}{\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1),\sigma \rightsquigarrow_{\mathsf{P}} \mathsf{v},\sigma''[\mathtt{this}{\mapsto}\sigma(\mathtt{this}),\mathsf{x}{\mapsto}\sigma(\mathsf{x})]}$$

For re-classification expressions, $\mathtt{this}{\Downarrow}\mathsf{d}$, we find the address of `this`, 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 $\mathcal{R}(\mathsf{P},\mathsf{c})$ denotes the least superclass of c which is not a state class: If c is a state class, then $\mathcal{R}(\mathsf{P},\mathsf{c})$ is its unique root superclass, otherwise $\mathcal{R}(\mathsf{P},\mathsf{c}){=}\ \mathsf{c}$. For example, $\mathcal{R}(\mathsf{P}_{\mathsf{st}},\mathsf{NonEmptyStack}) = \mathsf{Stack}$, and $\mathcal{R}(\mathsf{P}_{\mathsf{st}},\mathsf{StackException}) = \mathsf{StackException}$. Moreover $\mathcal{F}s(\mathsf{P},\mathsf{c})$ denotes the set of fields defined in class c, $\sigma(\iota)(\mathsf{f})$ the value of the field f in the object at address ι, and $\mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f})$ the type of field f in class c.

$$\frac{\begin{array}{l}\sigma(\mathtt{this}) = \iota\\ \sigma(\iota) = [[\ldots]]^{\mathsf{c}}\\ \mathcal{F}s(\mathsf{P},\mathcal{R}(\mathsf{P},\mathsf{c})) = \{\mathsf{f}_1,\ldots,\mathsf{f}_{\mathsf{r}}\}\\ \forall \mathsf{l}{\in}\,1,\ldots,\mathsf{r}: \quad \mathsf{v}_{\mathsf{l}} = \sigma(\iota)(\mathsf{f}_{\mathsf{l}})\\ \mathcal{F}s(\mathsf{P},\mathsf{d}) \setminus \{\mathsf{f}_1,\ldots,\mathsf{f}_{\mathsf{r}}\} = \{\mathsf{f}_{\mathsf{r}+1},\ldots,\mathsf{f}_{\mathsf{r}+\mathsf{q}}\}\\ \forall \mathsf{l}{\in}\,\mathsf{r}+1,\ldots,\mathsf{r}+\mathsf{q}: \quad \mathsf{v}_{\mathsf{l}} \text{ initial for } \mathcal{F}(\mathsf{P},\mathsf{d},\mathsf{f}_{\mathsf{l}})\end{array}}{\mathtt{this}{\Downarrow}\mathsf{d},\sigma \rightsquigarrow_{\mathsf{P}} \iota,\sigma[\iota{\mapsto}[[\mathsf{f}_1:\mathsf{v}_1,\ldots,\mathsf{f}_{\mathsf{r}+\mathsf{q}}:\mathsf{v}_{\mathsf{r}+\mathsf{q}}]]^{\mathsf{d}}]}$$

Take for instance program P_{st} from Figure 1. For a store σ_1, with $\sigma_1(\mathsf{s}) = \iota$, and $\sigma_1(\iota) = [[\mathsf{capacity}:\ 100,\ \mathsf{a}:\ \{3\},\ \mathsf{t}:\ 0]]^{\mathsf{NonEmptyStack}}$ we have $\mathsf{s.pop}(),\sigma_1 \rightsquigarrow_{\mathsf{P}_{\mathsf{st}}} \iota,\sigma_2$ where $\sigma_2 = \sigma_1[\iota{\mapsto}[[\mathsf{capacity}:\ 100]]^{\mathsf{EmptyStack}}]$ *i.e.* we obtain an object of class $\mathsf{EmptyStack}$ with unmodified $\mathsf{capacity}$.

Note that the rule for re-classification uses the types of the fields to initialize the fields, as the object creation does. In a well-typed program we always have $\mathcal{R}(\mathsf{P},\mathsf{c}) = \mathcal{R}(\mathsf{P},\mathsf{d})$ (and both c and d are state classes). This implies that re-classification depends only on the target class d, not on the class c of the receiver. Therefore, a compiler could fold the type information into the code, by generating specific re-classification code for each state class.

3.3 Typing

Widening, environments, effects It is useful to define some assertions: $\mathsf{P} \vdash \mathsf{c}\ \diamond_s$ means that c is a state class, $\mathsf{P} \vdash \mathsf{c}\ \diamond_r$ means that c is a root class, $\mathsf{P} \vdash \mathsf{c}\ \diamond_{nsr}$ means that c is a non-state, non-root class, $\mathsf{P} \vdash \mathsf{c}\ \diamond_c$ means that c is any class, $\mathsf{P} \vdash \mathsf{t}\ \diamond_{vt}$ means that t is a variable type *i.e.* either **bool**, or a non-state, non-root-class, $\mathsf{P} \vdash \mathsf{t}\ \diamond_t$ means that t is a

[5] We restore the references, but not the contents: thus, after a method call the receiver is the same, but any side effects caused by execution of the method body survive after the call.

type, *i.e.* any class or **bool**. Finally, $\mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}'$ means that type t' widens type t, *i.e.* t is a subclass of, or identical with, t'. In our example, $\mathsf{P_{st}} \vdash \mathsf{Stack} \diamond_r$ and $\mathsf{P_{st}} \vdash \mathsf{Stack} \diamond_{vt}$, and $\mathsf{P_{st}} \vdash \mathsf{EmptyStack} \diamond_s$, but $\mathsf{P_{st}} \nvdash \mathsf{EmptyStack} \diamond_{vt}$.

Environments, Γ, map the parameter x to variable types, and the receiver `this`to classes. Lookup, $\Gamma(\mathsf{id})$, update, $\Gamma[\mathsf{id} \mapsto \mathsf{t}]$, and well-formedness, $\mathsf{P} \vdash \Gamma \diamond$, have the usual meaning.

An effect, ϕ, is a set $\{ \mathsf{c_1}, ..., \mathsf{c_n} \}$ of root classes; it means that any object of a state subclass of $\mathsf{c_i}$ may be re-classified to any state subclass of $\mathsf{c_i}$. The empty effect, $\{\ \}$, guarantees that no object is re-classified. Effects are well formed, *i.e.* $\mathsf{P} \vdash \{ \mathsf{c_1}, ..., \mathsf{c_n} \} \diamond$, iff $\mathsf{c_1}$,...,$\mathsf{c_n}$ are distinct root classes. Thus, $\mathsf{P} \vdash \{ \mathsf{c_1}, ..., \mathsf{c_n} \} \diamond$ implies that $\mathsf{c_i}$ are not subclasses of each other.

Typing specialities Before discussing the typing rules, we look at some examples.

The type of `this` may change within a method body. In Example (4) `this` in method m

```
root class  A { }
state class  B  extends  A { bool j; }
state class  C  extends  A {
   bool i;
   bool m(){ A }{
      this.i := false;       // type correct, this is currently a C                (4)
      this.j := false;       //  type incorrect, this is currently a C
      this⇓B;
      this.i := false;       //  type incorrect, this is currently a B
      this.j := false       /*  type correct, this is currently a B  */ }
}
```

has type C before the re-classification, and it has type B afterwards. Changes to the type of `this` are caused either by explicit re-classification, as in Example (4), or by potential, indirect re-classification, as in methods h and k of Example (5).

If in method h, in (5), before aD.f() the field aD.anA happened to be an alias of the receiver, `this`, then `this` would be re-classified to B. In order to capture such potential re-classifications, each method declares as its effect the set of root classes of objects that may be re-classified through its execution. In our case, f has effect $\{ \mathsf{A} \}$. Therefore, after the call aD.f() the type of `this` is A, *i.e.* the application of the effect $\{ \mathsf{A} \}$ to class C.

Thus, typing an expression e, in the context of program P, and environment Γ involves three components, namely

$$\mathsf{P}, \Gamma \ \vdash \ \mathsf{e} \ : \ \mathsf{t} \ \| \ \mathsf{c} \ \| \ \phi$$

where t is the type of the value returned by evaluation of e, the class c is the class of `this` after evaluation of e, and ϕ conservatively estimates the re-classification effect of the evaluation of e on objects.

For example, let $\mathsf{P_4}$ and $\mathsf{P_5}$ be the programs from examples (4), and (5), and environments, Γ_0, Γ_1, Γ_2, with Γ_0(`this`)=C, Γ_1(`this`)=B, and Γ_2(`this`)=A. At the beginning of the body of m in (4), we have: $\mathsf{P_4}, \Gamma_0 \ \vdash$ `this.i` : **bool** $\|$ C $\|$ $\{\ \}$; then, the re-classification is typed as $\mathsf{P_4}, \Gamma_0 \ \vdash$ `this⇓B` : B $\|$ B $\|$ $\{\ \}$; therefore, after the re-classification, we use environment Γ_1, with $\mathsf{P_4}, \Gamma_1 \ \vdash$ `this.j` : **bool** $\|$ B $\|$ $\{\ \}$. The first expression of h in (5) is typed $\mathsf{P_5}, \Gamma_0 \ \vdash$ aD.f() : **bool** $\|$ A $\|$ $\{ \mathsf{A} \}$, and therefore the next term, `this.m(true)`, is checked in environment Γ_2, where it is type incorrect.

```
root class A { bool g(){A}{ true }   }
state class B extends A { }
state class C extends A {
  bool g(){A} { this⇓B; true }
  bool m(bool x){ }{ x }
  bool h(D aD){A}{
    this.m(true);                 // type correct
    aD.f();                       // might re-classify this
    this.m(true)                  /* type error */  }
  bool k(D aD){A}{                                                  (5)
    this.m(true);                 // type correct
    this.m(aD.f())                /* type error */  }
}

class D {
  A anA;
  bool f(){A}{ anA.g() }
}
```

The point from which effects modify the type of `this` is important. In method calls, the argument may affect the receiver. In example (5), in method k the first call of the method m is type correct, but the second is not. Namely, evaluation of the argument, aD.f(), may re-classify objects of subclasses of A, and therefore might re-classify `this`. Thus, the effect of the argument must be taken into account when looking-up the method. Because P_5, $\Gamma_0 \vdash$ `this` : C ‖ C ‖ { }, and P_5, $\Gamma_0 \vdash$ aD.f() : **bool** ‖ A ‖ {A}, we look-up the method m in class A; none is found, and so this.m(aD.f()) is type incorrect in environment Γ_0.

Typing Rules The typing rules are given in Figure 3, where we use look-up functions. In particular the functions $\mathcal{FD}(\mathsf{P},\mathsf{c},\mathsf{f})$, $\mathcal{MD}(\mathsf{P},\mathsf{c},\mathsf{m})$ search for fields and methods only in class c itself, the functions $\mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f})$, $\mathcal{M}(\mathsf{P},\mathsf{c},\mathsf{m})$ go through the class hierarchy.

We only discuss the rules for re-classification and for method call.

The re-classification this⇓c is type correct if c, the target of the re-classification, is a state class, and if c and the class of `this` before the mutation (the class Γ(`this`)) are subclasses of the same root class.

$$\frac{\mathsf{P} \vdash \mathsf{c} \diamond_s \qquad \mathcal{R}(\mathsf{P},\mathsf{c}) = \mathcal{R}(\mathsf{P},\Gamma(\texttt{this}))}{\mathsf{P},\Gamma \vdash \texttt{this}\Downarrow\mathsf{c} : \mathsf{c} \,\|\, \mathsf{c} \,\|\, \{\mathcal{R}(\mathsf{P},\mathsf{c})\}}$$

Consider method calls, $\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1)$. The evaluation of the argument e_1 may modify the class of the object e_0, as shown in example (5). This could happen if a superclass of the original class of e_0 is among the effects of e_1. (Existence of such a class implies uniqueness, since effects are sets of root classes.) The definition of m has to be found in the new class of the object e_0. For this purpose, we define the application of effects to classes:

$$\{\mathsf{c}_1,\ldots,\mathsf{c}_n\}@_{\mathsf{P}}\mathsf{c} = \begin{cases} \mathsf{c}_i & \text{if } \mathcal{R}(\mathsf{P},\mathsf{c}) = \mathsf{c}_i \text{ for some } i \in 1,\ldots,n \\ \mathsf{c} & \text{otherwise.} \end{cases}$$

For example, $\{\mathsf{Stack}\}@_{\mathsf{P}_{st}}\mathsf{NonEmptyStack} = \mathsf{Stack}$, and $\{\mathsf{Stack}\}@_{\mathsf{P}_{st}}\mathsf{Object} = \mathsf{Object}$.

For method call we lookup the definition of method m in the class obtained by applying the effect of the argument to the class of the receiver (which in general is not `this`):

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \;\vdash\; \mathsf{e}_0 \;:\; \mathsf{c} \parallel \mathsf{c}_0 \parallel \phi_0 \\ \mathsf{P},\Gamma[\mathtt{this}{\mapsto}\mathsf{c}_0] \;\vdash\; \mathsf{e}_1 \;:\; \mathsf{t}_1' \parallel \mathsf{c}_1 \parallel \phi_1 \\ \mathcal{M}(\mathsf{P},\phi_1@_{\mathsf{P}}\mathsf{c},\mathsf{m}) \;=\; \mathsf{t}\;\mathsf{m}(\mathsf{t}_1\;\mathsf{x})\;\phi\;\{\;\ldots\;\} \\ \mathsf{P} \vdash \mathsf{t}_1' \leq \mathsf{t}_1\end{array}}{\mathsf{P},\Gamma \;\vdash\; \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1) \;:\; \mathsf{t} \parallel \phi@_{\mathsf{P}}\mathsf{c}_1 \parallel \phi \cup \phi_0 \cup \phi_1}$$

Well-formed Programs A program is well formed (*i.e.* $\vdash \mathsf{P}\ \diamond$) if all its classes are well-formed (*i.e.* $\mathsf{P} \vdash \mathsf{c}\ \diamond$): Methods may override superclass methods only if they have the same name, argument, and result type, and their effect is a subset of that of the overridden method. Method bodies must be well formed, return a value appropriate for the method signature, and their effect must be a subset of that in the signature. See Figure 3, where $\mathcal{C}(\mathsf{P},\mathsf{c})$ returns the definition of class c in program P.

Soundness Figure 2 introduces agreement notions between programs, stores, and values:

- $\mathsf{P},\sigma \vdash \mathsf{v} \lhd \mathsf{t}$ means that value v agrees with type t in context of program P and store σ;
- $\mathsf{P},\Gamma \vdash \sigma\ \diamond$ means that store σ agrees with environment Γ and program P.

$$\frac{}{\mathsf{P},\sigma \vdash \mathtt{true} \lhd \mathbf{bool}} \qquad \frac{}{\mathsf{P},\sigma \vdash \mathtt{false} \lhd \mathbf{bool}} \qquad \frac{\mathsf{P} \vdash \mathsf{t}\ \diamond_c}{\mathsf{P},\sigma \vdash \mathtt{null} \lhd \mathsf{t}}$$

$$\frac{\sigma(\iota) = [[\ldots]]^{\mathsf{c}} \qquad \mathsf{P} \vdash \mathsf{c} \leq \mathsf{t} \qquad \forall \mathsf{f} \in \mathcal{F}s(\mathsf{P},\mathsf{c}) :\ \mathsf{P},\sigma \vdash \sigma(\iota)(\mathsf{f}) \lhd \mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f})}{\mathsf{P},\sigma \vdash \iota \lhd \mathsf{t}} \qquad \frac{\begin{array}{l}\sigma(\iota) = [[\ldots]]^{\mathsf{c}} \implies \mathsf{P},\sigma \vdash \iota \lhd \mathsf{c}, \quad \text{for all addresses } \iota, \\ \mathsf{P},\sigma \vdash \sigma(\mathtt{this}) \lhd \Gamma(\mathtt{this}), \qquad \mathsf{P},\sigma \vdash \sigma(\mathsf{x}) \lhd \Gamma(\mathsf{x})\end{array}}{\mathsf{P},\Gamma \;\vdash\; \sigma\diamond}$$

Fig. 2. Agreement between programs, stores, and values

The type system is sound in the sense that a converging well-typed expression returns `nullPntrExc`, or a value which agrees with the expression's type; but is *never* stuck. The resulting state agrees with the program and the environment (taking the effect into account):

Theorem 1 (Type Soundness) *For well-formed program* P*, environment* Γ*, expression* e*, and type* t*, such that*

$$\mathsf{P},\Gamma \;\vdash\; \mathsf{e} \;:\; \mathsf{t} \parallel \mathsf{c} \parallel \phi$$

if $\mathsf{P},\Gamma \;\vdash\; \sigma\ \diamond$*, and* e,σ *converges then*

- $\mathsf{e},\sigma \rightsquigarrow_{\mathsf{P}} \mathsf{v},\sigma', \quad \mathsf{P},\sigma' \vdash \mathsf{v} \lhd \mathsf{t}, \quad$ *and* $\quad \mathsf{P},\Gamma[\mathtt{this}{\mapsto}\mathsf{c}] \vdash \sigma'\ \diamond$,
  or
- $\mathsf{e},\sigma \rightsquigarrow_{\mathsf{P}} \mathtt{nullPntrExc},\sigma', \quad$ *and* $\quad \mathsf{P},\Gamma[\mathtt{this}{\mapsto}(\phi@_{\mathsf{P}}\Gamma(\mathtt{this}))] \vdash \sigma'\ \diamond$.

Proof outline We introduce a notion of agreement between states and effects, requiring receivers to remain the same, and re-classifications to be between subclasses of the effect:

$\mathsf{P},\phi \vdash \sigma \lhd \sigma'$ iff
- $\sigma(\mathtt{this}) = \sigma'(\mathtt{this})$, and,
- $\sigma(\iota) = [[\ldots]]^{\mathsf{c}} \implies \sigma'(\iota) = [[\ldots]]^{\mathsf{c}'},\ \phi@_{\mathsf{P}}\mathsf{c} = \phi@_{\mathsf{P}}\mathsf{c}'$.

With this notion of agreement we prove the following, stronger, theorem:

If $\mathsf{P}, \Gamma \vdash \mathtt{e} : \mathtt{t} \parallel \mathtt{c} \parallel \phi$, and $\mathsf{P}, \Gamma \vdash \sigma \diamond$, and $\mathtt{e}, \sigma$ converges, then

- $\mathtt{e}, \sigma \overset{}{\rightsquigarrow}_{\mathsf{P}} \mathtt{v}, \sigma'$, $\quad \mathsf{P}, \sigma' \vdash \mathtt{v} \lhd \mathtt{t}$, $\quad \mathsf{P}, \phi \vdash \sigma \lhd \sigma'$, $\quad \mathsf{P}, \Gamma[\mathtt{this} \mapsto \mathtt{c}] \vdash \sigma' \diamond$,

  or
- $\mathtt{e}, \sigma \overset{}{\rightsquigarrow}_{\mathsf{P}} \mathtt{nullPntrExc}, \sigma'$, $\quad \mathsf{P}, \phi \vdash \sigma \lhd \sigma'$, $\quad \mathsf{P}, \Gamma[\mathtt{this} \mapsto (\phi @_{\mathsf{P}} \Gamma(\mathtt{this}))] \vdash \sigma' \diamond$.

The proof is by induction on the derivations of the operational semantics. The full proof can be found at http://www.di.unito.it/~damiani/papers/dor.html. □

Remark 2. As far as divergent expressions go, the theorem does not say anything. However, the operational semantics forces convergence for standard typing errors or access to members undefined for an object. Therefore, Theorem 1 suffices to ensure that for a well-typed expression such errors will not occur.

Remark 3. The weaker guarantee of well-formedness for the resulting store σ' in the second case of Theorem 1, is due to the fact that the interruption of execution of $\mathtt{e}$ might prevent setting the type of `this` to $\mathtt{c}$. For instance, for program P_0, state classes $\mathtt{d}'$, $\mathtt{d}''$, which are not subclasses of each other, and $\mathtt{d}$ their root superclass, σ_0, Γ_0 and $\mathtt{e}_0$ with $\sigma_0(\mathtt{this}) = [[\cdots]]^{\mathtt{d}'}$ and $\Gamma_0(\mathtt{this}) = \mathtt{d}'$, and $\mathtt{e}_0 = \mathtt{null.f}; \mathtt{this} \Downarrow \mathtt{d}''$, typing produces:

$$\mathsf{P}_0, \Gamma \vdash \mathtt{e}_0 : \mathtt{d}'' \parallel \mathtt{d}'' \parallel \{\mathtt{d}\},$$

whereas execution produces:

$$\mathtt{e}_0, \sigma_0 \overset{}{\rightsquigarrow}_{\mathsf{P}_0} \mathtt{nullPntrExc}, \sigma_0.$$

In σ_0 the receiver `this` is bound to an object of class $\mathtt{d}'$. So, $\mathsf{P}_0, \Gamma[\mathtt{this} \mapsto \mathtt{d}''] \not\vdash \sigma_0 \diamond$. However, $\{\mathtt{d}\}@_{\mathsf{P}_0}\Gamma_0(\mathtt{this}) = \mathtt{d}$, and $\mathsf{P}_0 \vdash \mathtt{d}' \sqsubseteq \mathtt{d}$. Thus, $\mathsf{P}_0, \Gamma_0[\mathtt{this} \mapsto \{\mathtt{d}\}@_{\mathsf{P}}\Gamma_0(\mathtt{this})] \vdash \sigma_0 \diamond$ holds.

4 Related work

Most foundational work is based on functional object-based languages. In [1] method overriding models field update and delegation. In [10] method extensions represent class inheritance, while [3, 7, 18, 11, 19] enhance the above representation by introducing a limited form of method subtyping. These calculi deal with questions of width-subtyping over breadth-subtyping, the use of MyType, method extension and overriding; they were primarily developed as a means of understanding inheritance and delegation.

Object extension in these calculi can be seen as the promotion of an object of class $\mathtt{c}$ to an object of a subclass of $\mathtt{c}$. In [11, 18, 7, 13, 3], unrestricted subtyping followed by object expansion might cause messageNotUnderstood errors, and so type soundness is recovered by imposing certain restrictions on the use of subtyping, with the consequence that an object cannot be promoted to a superclass and then to the original subclass.

For databases, [2] suggests multiple most specific classes, thus in a way allowing multiple inheritance, while [13] allows objects to accumulate different roles in a functional setting. They model non-exclusive roles (*e.g.* female and professor), whereas we model objects changing mutually exclusive classes (*e.g.* opened window versus iconified window).

Refinement types in functional languages distinguish cases through subtypes, see [12]. The main questions in [12] are type inference, and establishing that functions are well defined, that is they cover all possible cases. Side-effects are not considered, therefore questions like aliasing that are central to our development do not arise.

Predicate classes [5, 9], on an imperative setting, suggest multi-method dispatch depending on predicates on the receiver and arguments. Code is broken down on a per-function basis, while *Fickle* follows the mainstream, whereby code is broken down on a per-class basis. Also, in [5] the term "re-classifcation" denotes changes in attribute values which imply

changes in predicates when calculated next. Thus, re-classification in [5, 9] is implicit and lazy, whereas in *Fickle* it is explicit and eager. In [5, 9] different methods may dispatch depending on different predicates, *e.g.* insert depends on priority vs. precedence lists, whereas print depends on empty vs. non-empty lists. This is not possible in *Fickle*, unless perhaps, extended with multiple inheritance. Finally, [5, 9] raise the question of disjointness and completeness of predicates (unambiguous and complete).

Similarly, for single method dispatch, in [22] classes have "modes" representing different states, *e.g.* opened vs. iconified window. Wide classes [21] are the nearest to our approach, and allow an object to be temporarily "widened" or "shrunk". However they differ from *Fickle*, by dropping the requirement for a strong type system, and requiring run-time tests for the presence of fields. (Anyway the aim of wide classes was to have a better memory use in presence of changes of object structures.)

Fickle is the successor of an earlier proposal, *Fickle*–99 [8], which addresses the same requirement. *Fickle* improves *Fickle*–99 in at least two respects. Firstly, in *Fickle*–99 we needed to prevent objects from mutating while executing a method, and achieved this either through run-time locks or through an effect system. Secondly, in *Fickle*–99 we had to distinguish three kinds of methods, two kinds of objects and two kinds of types.

5 Design Alternatives

Our aim was to develop language features supporting re-classification of objects in an imperative setting, allowing aliasing. Thus, we fixed the operational semantics very early, but the design of a safe type system was not straightforward. The main challenges were:

1. The type of `this` inside method bodies containing re-classifications; – *c.f.* Example (4), Section 3.3.
2. Re-classification of an aliased object may remove members, which the object needs in another context– *c.f.* Example (3), Section 2.
3. Re-classification of an object while it executes a method which uses members removed by a re-classification further down the call stack – *c.f.* Example (5), Section 3.3.

We have considered, and experimented with several ideas:

1. The type of `this` changes after re-classifications; we express that through the second component of our typing scheme.
2. We considered several solutions, and have chosen (e):
   (a) Check the existence of members at run-time, as in [21]; but this is type-unsafe.
   (b) An object should have all members for all possible state subclasses of its root superclass, as in [9]. Although type safe, this does not allow compact representations as required in [21], and does not express our intention of exclusive cases.
   (c) Require all state subclasses of a root class to have exactly the same members, and differ in the method bodies only. However, this requirement is too strong, *e.g.* does not hold for empty and non-empty stacks.
   (d) In *Fickle*–99, we avoid the aliasing introduced through line 2. in Example (3). Types are either non-state, non-root classes, or sets of state classes. Subtyping for such sets of state classes is only the identity. Accessing a member of an expression with type a set of state classes is only legal if all state classes define this member.
   (e) In *Fickle*, we forbid the use of state classes as types, except for the type of `this`. Thus state classes may have different members, but all state subclasses of the same root class offer the same interface to all their clients.

3.

(a)-(c) With any of the approaches described in 2(a), 2(b) or 2(c), the problem would not arise; but we have rejected these solutions in 2.

(d) In *Fickle*–99 we "lock" an object of a state class when it starts executing a method, and "unlock" when it finishes. Attempting to re-classify a locked object throws an exception; *e.g.* Example (5) could throw such an exception. This is too restrictive, and has the draw-back that it allows run-time errors.

(e) In *Fickle* the type system ensures against the problem; the effects from any called methods are applied to the type of `this`; therefore, after a call which may modify the receiver, the type of `this` will be the root superclass, and so, access to state class members will be type incorrect.

6 Conclusions

Fickle is the outcome of several designs and successive improvements. We are now satisfied that the suggested approach is useful and usable. In the process, we also developed an interesting typing scheme, where typing an expression affects the environment in which the following or enclosing expressions are typed.

We are working on an implementation of *Fickle* through a Java preprocessor [15]. Type correct *Fickle* programs are mapped into equivalent Java programs, where root classes are represented by wrapper classes, containing a field value, which points to an object of one of its state subclasses. Method calls are forwarded from the wrapper object to its value field, and re-classifications are implemented by overwriting the value field. Also, we have several paper examples, *e.g.* accounts, linked lists, adventure games, and cases delineating the typing rules [16].

In a production compiler one can avoid the wrapper object and the indirection in the method dispatch, provided that the maximal size of state subclasses of any given root is known[6]: Notably, the constraint that the target and source of re-classification have a common root superclass allows the standard, efficient implementation of method call, where we lookup through an offset into the method dispatch table of the receiver. The fact that sources and targets of re-classifications have the same maximal size allows to implement re-classification through simple in-place overwriting of the source object.

Further work includes finishing the implementation, the incorporation of *Fickle* into a full language, the refinement of the effect system *e.g.* through data-flow analysis techniques, the incorporation of myType, multiple inheritance, the distinction of subclassing from subtyping, and the modelling of irreversible re-classifications (*e.g.* pupa to butterfly).

Acknowledgements

Fickle has benefited from constructive criticism on *Fickle*–99 from Walt Hill, Viviana Bono, Luca Cardelli, Andrew Kennedy, Giorgio Ghelli, and anonymous POPL'00 reviewers. Ross Jarman, and the anonymous FOOL'01 referees gave useful feedback on our current work.

[6] Restrictions on possible subclasses can be found in several systems, *e.g.* in [6].

$$\frac{\mathsf{P} \vdash \mathsf{c} \diamond_s \qquad \mathcal{R}(\mathsf{P},\mathsf{c}) = \mathcal{R}(\mathsf{P},\Gamma(\mathtt{this}))}{\mathsf{P},\Gamma \vdash \mathtt{this}\Downarrow\mathsf{c} : \mathsf{c} \parallel \mathsf{c} \parallel \{\mathcal{R}(\mathsf{P},\mathsf{c})\}}$$

$$\frac{\mathsf{P},\Gamma \vdash \mathsf{e}_0 : \mathsf{c} \parallel \mathsf{c}_0 \parallel \phi_0 \qquad \mathsf{P},\Gamma[\mathtt{this}\mapsto\mathsf{c}_0] \vdash \mathsf{e}_1 : \mathsf{t}_1' \parallel \mathsf{c}_1 \parallel \phi_1 \qquad \mathcal{M}(\mathsf{P},\phi_1@_{\mathsf{P}}\mathsf{c},\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x})\ \phi\ \{\ \ldots\ \} \qquad \mathsf{P} \vdash \mathsf{t}_1' \leq \mathsf{t}_1}{\mathsf{P},\Gamma \vdash \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1) : \mathsf{t} \parallel \phi@_{\mathsf{P}}\mathsf{c}_1 \parallel \phi \cup \phi_0 \cup \phi_1}$$

$$\frac{\mathsf{P} \vdash \Gamma \diamond}{\mathsf{P},\Gamma \vdash \mathtt{true} : \mathtt{bool} \parallel \Gamma(\mathtt{this}) \parallel \{\} \qquad \mathsf{P},\Gamma \vdash \mathtt{false} : \mathtt{bool} \parallel \Gamma(\mathtt{this}) \parallel \{\} \qquad \mathsf{P},\Gamma \vdash \mathsf{x} : \Gamma(\mathsf{x}) \parallel \Gamma(\mathtt{this}) \parallel \{\} \qquad \mathsf{P},\Gamma \vdash \mathtt{this} : \Gamma(\mathtt{this}) \parallel \Gamma(\mathtt{this}) \parallel \{\}}$$

$$\frac{\mathsf{P} \vdash \Gamma \diamond \qquad \mathsf{P} \vdash \mathsf{c} \diamond_c}{\mathsf{P},\Gamma \vdash \mathtt{null} : \mathsf{c} \parallel \Gamma(\mathtt{this}) \parallel \{\} \qquad \mathsf{P},\Gamma \vdash \mathbf{new}\ \mathsf{c} : \mathsf{c} \parallel \Gamma(\mathtt{this}) \parallel \{\}}$$

$$\frac{\mathsf{P},\Gamma \vdash \mathsf{e} : \mathsf{c} \parallel \mathsf{c}' \parallel \phi \qquad \mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f}) = \mathsf{t}}{\mathsf{P},\Gamma \vdash \mathsf{e}.\mathsf{f} : \mathsf{t} \parallel \mathsf{c}' \parallel \phi}$$

$$\frac{\mathsf{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \mathsf{c} \parallel \phi \qquad \mathsf{P},\Gamma[\mathtt{this}\mapsto\mathsf{c}] \vdash \mathsf{e}' : \mathsf{t}' \parallel \mathsf{c}' \parallel \phi'}{\mathsf{P},\Gamma \vdash \mathsf{e};\ \mathsf{e}' : \mathsf{t}' \parallel \mathsf{c}' \parallel \phi \cup \phi'}$$

$$\frac{\mathsf{P},\Gamma \vdash \mathsf{x} : \mathsf{t} \parallel \mathsf{c} \parallel \{\} \qquad \mathsf{P},\Gamma[\mathtt{this}\mapsto\mathsf{c}] \vdash \mathsf{e} : \mathsf{t}' \parallel \mathsf{c}' \parallel \phi' \qquad \mathsf{P} \vdash \mathsf{t}' \leq \mathsf{t}}{\mathsf{P},\Gamma \vdash \mathsf{x}{:=}\mathsf{e} : \mathsf{t}' \parallel \mathsf{c}' \parallel \phi'}$$

$$\frac{\mathsf{P},\Gamma \vdash \mathsf{e} : \mathsf{c}' \parallel \mathsf{c}'' \parallel \phi \qquad \mathsf{P},\Gamma[\mathtt{this}\mapsto\mathsf{c}''] \vdash \mathsf{e}' : \mathsf{t} \parallel \mathsf{c} \parallel \phi' \qquad \mathcal{F}(\mathsf{P},\phi'@_{\mathsf{P}}\mathsf{c}',\mathsf{f}) = \mathsf{t}' \qquad \mathsf{P} \vdash \mathsf{t} \leq \mathsf{t}'}{\mathsf{P},\Gamma \vdash \mathsf{e}.\mathsf{f}{:=}\mathsf{e}' : \mathsf{t} \parallel \mathsf{c} \parallel \phi \cup \phi'}$$

$$\frac{\mathsf{P},\Gamma \vdash \mathsf{e} : \mathtt{bool} \parallel \mathsf{c} \parallel \phi \qquad \mathsf{P},\Gamma[\mathtt{this}\mapsto\mathsf{c}] \vdash \mathsf{e}_1 : \mathsf{t}_1 \parallel \mathsf{c}_1 \parallel \phi_1 \qquad \mathsf{P},\Gamma[\mathtt{this}\mapsto\mathsf{c}] \vdash \mathsf{e}_2 : \mathsf{t}_2 \parallel \mathsf{c}_2 \parallel \phi_2 \qquad \mathsf{P} \vdash \mathsf{t}_i \leq \mathsf{t} \text{ for } \mathsf{i} \in 1,2}{\mathsf{P},\Gamma \vdash \mathbf{if}\ \mathsf{e}\ \ \mathbf{then}\ \ \mathsf{e}_1\ \ \mathbf{else}\ \mathsf{e}_2 : \mathsf{t} \parallel \mathsf{c}_1 \sqcup_{\mathsf{P}} \mathsf{c}_2 \parallel \phi \cup \phi_1 \cup \phi_2}$$

$$\frac{\begin{array}{l}\vdash \mathsf{P} \diamond_a \\ \mathcal{C}(\mathsf{P},\mathsf{c}) = [\mathbf{root} \mid \mathbf{state}]\mathbf{class}\ \mathsf{c}\ \mathbf{extends}\ \mathsf{c}'\ \{\ldots\} \\ \forall \mathsf{f}: \ \mathcal{FD}(\mathsf{P},\mathsf{c},\mathsf{f}) = \mathsf{t}_0 \implies \mathsf{P} \vdash \mathsf{t}_0 \diamond_{vt} \quad \text{and} \quad \mathcal{F}(\mathsf{P},\mathsf{c}',\mathsf{f}) = \mathcal{U}df \\ \forall \mathsf{m}: \ \mathcal{MD}(\mathsf{P},\mathsf{c},\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x})\ \phi\ \{\ \mathsf{e}\ \} \implies \\ \qquad \mathsf{P} \vdash \mathsf{t} \diamond_{vt} \\ \qquad \mathsf{P} \vdash \mathsf{t}_1 \diamond_{vt} \\ \qquad \mathsf{P} \vdash \phi \diamond \\ \qquad \mathsf{P},\mathsf{t}_1\ \mathsf{x},\mathsf{c}\ \mathtt{this} \vdash \mathsf{e} : \mathsf{t}' \parallel \mathsf{c}'' \parallel \phi' \\ \qquad \mathsf{P} \vdash \mathsf{t}' \leq \mathsf{t} \\ \qquad \phi' \subseteq \phi \\ \qquad \mathcal{R}(\mathsf{P},\mathsf{c}) = \mathcal{R}(\mathsf{P},\mathsf{c}'') \\ \qquad \mathcal{M}(\mathsf{P},\mathsf{c}',\mathsf{m}) = \mathcal{U}df \quad \text{or} \quad (\mathcal{M}(\mathsf{P},\mathsf{c}',\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x})\ \phi''\ \{\ \mathsf{e}'\ \} \quad \text{and} \quad \phi \subseteq \phi'')\end{array}}{\mathsf{P} \vdash \mathsf{c} \diamond}$$

$$\frac{\forall \mathsf{c}: \ \mathcal{C}(\mathsf{P},\mathsf{c}) \neq \mathcal{U}df \implies \mathsf{P} \vdash \mathsf{c} \diamond}{\vdash \mathsf{P} \diamond}$$

Fig. 3. Typing rules for expressions and well-formed classes and programs

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In *ECOOP'95*, volume 952 of *LNCS*, pages 102–126. Springer, 1995.
3. V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraints for Incomplete Objects. In *CAAP'97*, volume 1214 of *LNCS*, pages 465–477. Springer, 1997.
4. P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for Strongly Typed Object Oriented Languages. In *OOPSLA'89*, pages 457–467. ACM press, 1989.
5. C. Chambers. Predicate Classes. In *ECOOP'93*, volume 707 of *LNCS*, pages 268–296. Springer, 1993.
6. C. Chambers and G. Leavens. Type Checking Modules for Multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, 1995.
7. P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *OOPSLA'98*, pages 166–178. ACM press, 1998.
8. S. Drossopoulou, M. Dezani-Ciancaglini, F. Damiani, and P. Giannini. Objects Dynamically Changing Class. Technical report, Imperial College, August 1999. Available from http://www.di.unito.it/~dezani/odcc.html.
9. M. D. Ernst, C. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *ECOOP'98*, volume 1445 of *LNCS*, pages 186–211. Springer, 1998.
10. K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. In *Nordic Journal of Computing 1(1)*, pages 3–37, 1994.
11. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *FCT'95*, volume 965 of *LNCS*, pages 42–61. Springer, 1995.
12. T. Freeman and F. Pfenning. Refinement types for ML. In *SIGPLAN '91*, pages 268–277. ACM Press, 1991.
13. G. Ghelli and D. Palmerini. Foundations of Extended Objets with Roles (*extended abstract*). In *FOOL'06*, 1999. Available from http://www.cs.williams.edu/~kim/FOOL/FOOL6.html.
14. W.L. Hürsch. Should Superclasses be Abstract? In *ECOOP'94*, volume 821 of *LNCS*, pages 12–31. Springer, 1994.
15. R. Jarman. Fickle: a Study in Objects, Imperial College, final year thesis, August 2000.
16. R. Jarman and S. Drossopoulou. Examples in Fickle. Available from http://www.di.unito.it/~damiani/papers/dor.html.
17. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL'88*, pages 47–57. ACM press, 1988.
18. D. Rémy. From Classes to Objects via Subtyping. In *ESOP'98*, volume 1381 of *LNCS*, pages 200–220. Springer, 1995.
19. J. C. Riecke and C. A. Stone. Privacy via Subsumption. In *FOOL'98*, 1998. Available from http://www.cs.williams.edu/~kim/FOOL/FOOL5.html.
20. T. Scheer and S. Pringle. Ten Practical Limitations of Object Orientation, November 1998. OOPSLA Poster Session, Available from http://www.acm.org/sigplan/oopsla/oopsla98/fp/posters/10.htm.
21. M. Serrano. Wide Classes. In *ECOOP'99*, volume 1628 of *LNCS*, pages 391–415. Springer, 1999.
22. A. Tailvasaari. Object Oriented Programming with Modes. *Journal of Object Oriented Programming*, pages 27–32, 1992.
23. J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.