Refined Effects for Unanticipated Object Re-classification: $\mathcal{F}ickle_3$*
(Extended Abstract)

Ferruccio Damiani[1], Sophia Drossopoulou[2], and Paola Giannini[3]

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

Abstract. In previous work on the language $\mathcal{F}ickle$ and its extension $\mathcal{F}ickle_{\rm II}$ Dezani and us introduced language features for object re-classification for imperative, typed, class-based, object-oriented languages.
In this paper we present the language $\mathcal{F}ickle_3$, which on one side refines $\mathcal{F}ickle_{\rm II}$ with more expressive effect annotations, and on the other eliminates the need to declare explicitly which are the classes of the objects that may be re-classified. Therefore, $\mathcal{F}ickle_3$ allows to correctly type meaningful programs which $\mathcal{F}ickle_{\rm II}$ rejects. Moreover, re-classification may be decided by the client of a class, allowing *unanticipated object re-classification*. As for $\mathcal{F}ickle_{\rm II}$, also the type and effect system for $\mathcal{F}ickle_3$ guarantees that, even though objects may be re-classified across classes with different members, they will never attempt to access non existing members.
The type and effect system of $\mathcal{F}ickle_3$ has some significant differences from the one of $\mathcal{F}ickle_{\rm II}$. In particular, besides the fact that intra-class type checking has to track the more refined effects, when a class is combined with other classes some additional inter-class checking is introduced.

1 Introduction

Re-classifiable objects support the changing of an object's behaviour by changing its class membership at runtime, see e.g. [2, 10, 8, 11]. In previous work on the language $\mathcal{F}ickle$ [5] and its extension $\mathcal{F}ickle_{\rm II}$ [6] Dezani and us introduced language features which allow objects to change class membership dynamically and showed how to combine these features with a strong type system. We based our approach on an imperative, class-based language, where classes are types and subclasses are subtypes, and where methods are defined inside classes and selected depending on the class of the object on which the method is invoked.

In this paper we present the language $\mathcal{F}ickle_3$, which on one side refines $\mathcal{F}ickle_{\rm II}$ with more expressive effect annotations, and on the other eliminates

* Work partially supported by IST-2001-33477 DART, MIUR Cofin '01 NAPOLI, and MIUR Cofin'02 McTati projects. The founding bodies are not responsible for any use that might be made of the results presented here.

the need to declare explicitly which are the classes of the objects that may be re-classified. We focus on the differences of the new proposal over $\mathcal{F}ickle_{\mathtt{II}}$. For comparison with other proposals in the literature we refer to [6].

$\mathcal{F}ickle_3$ is a Java-like language with a *re-classification* operation that changes the class membership of an object while preserving its identity. The basic problem in the design of the language is to have a sound type system, that is, a type system that, even in the presence of object re-classification, insures that no attempt is made at accessing non existing members. This is obtained by changing the type of `this` and of local identifiers that might point at a re-classified object in a method body. An object o of class c could also be pointed at by a field f of type d (of some other object o'), where d is a superclass of c. Therefore, to achieve type soundness, we must forbid re-classification from (a subclass of) d to a class which is not a subclass of d (otherwise subsequent accesses to members of o'.f existing in class d could fail). Classes for which this does not happen are said to be "respected" by re-classification and can be safely used as types for fields.

- In $\mathcal{F}ickle_{\mathtt{II}}$, this was guaranteed by explicitly marking classes which had to be respected by re-classification with the keywords **state** and **root**. *State* classes are the possible sources and targets of re-classifications, and *root* classes are the superclasses of state classes and declare all the members common to them.
- In $\mathcal{F}ickle_3$ the set of classes that are respected by re-classification is inferred by tracing all possible re-classifications in the program.

Re-classifications are traced by *effects*, see [9, 12], and methods are annotated with the effects that may be caused by the execution of their body.

- In $\mathcal{F}ickle_{\mathtt{II}}$ effects are just sets of root classes, $\{c_1, ..., c_n\}$, meaning that the method may perform any re-classification between two subclasses of a class in the set.
- In $\mathcal{F}ickle_3$ effects are sets of pairs, $\{ c_1 \Downarrow c'_1, ..., c_n \Downarrow c'_n \}$, meaning that the method may perform any re-classification from a subclass of c_i to a subclass c'_i.

$\mathcal{F}ickle_{\mathtt{II}}$-style effect annotations can be coded into $\mathcal{F}ickle_3$-style effect annotations, but not vice versa, and $\mathcal{F}ickle_3$ allows to correctly type meaningful programs which $\mathcal{F}ickle_{\mathtt{II}}$ rejects. Moreover, by eliminating the explicit marking of classes that must be respected by re-classification, we obtain a language that is syntactically simpler, and allows *unanticipated dynamic object replacement* (see e.g. [3]). However, this has a drawback: when linking a class c with an existing set of classes we have to check on one side that the effects of the methods of c respect the classes used as types of fields of the preexisting classes, and on the other that the effects of the methods of the preexisting classes respect the classes used as types of fields of c.

The operational semantics (which ignores effect annotations and types) of $\mathcal{F}ickle_3$ is essentially unchanged w.r.t. $\mathcal{F}ickle_{\mathrm{II}}$,[4] whereas in the type and effect system there are some significant differences. In particular, as previously mentioned, along with the intra-class checking we also need some inter-class checking.

This paper is organized as follows: In Section 2 we introduce $\mathcal{F}ickle_3$ informally using an example. In Section 3 we give syntax and operational semantics. In Section 4 we present the typing rules and state type soundness. Some technical definitions are listed in the appendix.

2 The Example of the Frog and of the Prince

In Figure 1 we give, by using a syntax similar to Java's, an example inspired by adventure games. The example is essentially the same example illustrated in Section 2 of [6]. Besides the differences in the syntax of the language, the only difference is that, in class Princess, we have replaced the method walk2 with the method walk3.

We define a class Player with subclasses Frog and Prince. When woken up, a frog inflates its pouch, while a prince swings his sword. When kissed, a frog turns into a prince; when cursed, a prince turns into a frog.

Annotations like $\{\ \}$, $\{\,\mathsf{Frog}{\Downarrow}\mathsf{Prince}\,\}$, and $\{\,\mathsf{Prince}{\Downarrow}\mathsf{Frog}\,\}$ before method bodies are called *effects*; expressions like Prince⇓Frog and Frog⇓Prince are called *atomic effects*. Effects list potential re-classifications that may be caused by the invocation of that method (each atomic effect represents a potential re-classification). Methods with the empty effect $\{\ \}$, *e.g.* wake, may not cause any re-classification. Methods with non-empty effects, *e.g.* kissed with effect $\{\,\mathsf{Frog}{\Downarrow}\mathsf{Prince}\,\}$, may re-classify objects from a subclass of the left-hand side of one of their atomic effects to a subclass of the right-hand side of that atomic effect; in our case from Frog to Prince. Such re-classifications may be caused by *re-classification expressions* (*e.g.* `this`⇓Prince in method kissed of class Frog, or mate2⇓Prince in method walk3 of class Princess), or by further method calls (*e.g.* mate.kissed () in method walk1 of class Princess).

The classes c such that an object belonging to some subclass of c may be re-classified to a class that is not a subclass of c *cannot* be used as type for fields; in our example Frog and Prince.

The method body of kissed in class Frog contains the re-classification expression `this`⇓Prince. At the beginning of the method the receiver is an object of class Frog, therefore it contains the fields brave and pouch, but not the field sword. After execution of `this`⇓Prince the receiver is of class Prince, and therefore sword can be selected, while pouch can not, and brave retains its value. This mechanism supports the transmission of some information from the object before the re-classification to the object after the re-classification.

[4] The only difference is that, in $\mathcal{F}ickle_3$, object re-classification preserves the fields of the least common superclass of the source and target of re-classification while, in $\mathcal{F}ickle_{\mathrm{II}}$, the field preserved are those of the root class.

```
abstract class Player extends Object{
  bool brave;
  abstract bool wake()  { } ;
  abstract Weapon kissed()  { Frog⇓Prince } ;
}
class Frog extends Player{
  Vocal pouch;
  bool wake() {  } {pouch.blow() ; brave}
  Weapon kissed() { Frog⇓Prince } {this⇓Prince; sword:= new Weapon}
}
class Prince extends Player{
  Weapon sword;
  bool wake() {  } {sword.swing(); brave}
  Weapon kissed() {  } {sword}
  Frog cursed() { Prince⇓Frog } {this⇓Frog; pouch:= new Vocal; this}
}
class Princess extends Object{
  bool walk1(Frog mate) { Frog⇓Prince } {mate.wake();  mate.kissed();  mate.wake()}
  Weapon walk3(Prince mate1, Frog mate2) { Frog⇓Prince }
    {mate2⇓Prince;  mate2.sword:= mate1.sword}
}
```

Fig. 1. Program P_{pl}- players with re-classifications in $\mathcal{F}ickle_3$

Consider the instructions in the method body of walk1 in class Princess:

```
1.  mate.wake();       // inflates pouch
2.  mate.kissed();                                    (1)
3.  mate.wake()        // swings sword
```

Suppose that the parameter mate is bound to a Frog object with field brave containing true. After line 2., the object is re-classified to Prince with the same value for brave. Therefore, the call of wake in line 1. selects the method from Frog, and inflates the pouch, while the call of wake in line 3. selects the method from Prince, and swings the sword.

Re-classification from class c_1 to class c_2 removes from the object all fields that are not defined in the least common superclass of c_1 and c_2, and adds the remaining fields of the target class. *E.g.* after line 2. in example (1) the object denoted by mate has a sword but not a pouch.

Consider now the instructions in the method body of walk3 in class Princess:

```
1.  mate2⇓Prince;
2.  mate2.sword:= mate1.sword                 (2)
```

Let the parameters mate1 and mate2 be bound to a Prince and to a Frog object, respectively. After line 1., the object pointed at by mate2 is re-classified to Prince and, in the left-hand side of line 2., field sword can be selected. Moreover, the object pointed at by mate1 is unchanged and, in the right-hand side of line 2, field sword can be selected. So the execution of this method is safe.

As a matter of fact in $\mathcal{F}ickle_{\mathrm{II}}$ [6] walk3 does not type-check because the effect inferred for the re-classification mate2$\Downarrow$Prince, in line 1, is $\{$Player $\}$, meaning that the type of all the local identifiers having as type a subclass of Player is changed to Player (except for mate2, whose type is changed to Prince). So in line 2 the variable mate1 is of class Player and mate1.sword gives type error. Note that mate2.sword type-checks both in $\mathcal{F}ickle_{\mathrm{II}}$ and $\mathcal{F}ickle_3$.

3 Syntax and Operational Semantics

The syntax of $\mathcal{F}ickle_3$ is given in Fig. 2.[5] We use standard extended BNF, where a [-] pair means optional, and A^* means zero or more repetitions of A. We follow the convention that non terminals appear as *nonTerm*, keywords appear as **keyword**, literals appear as `literal` and identifiers appear as identifier. We omit separators like "**;** " or "**,** " where they are obvious. Expressions are usually called e, e′, e_1 *etc.*, and values are usually called v, v′, v_1 *etc.*. By id, id′,*etc.*we will denote either `this` or a parameter name (x, x′,*etc.*). A program is a sequence of class definitions. Method declarations have the shape:

$$\mathsf{t}\ \mathsf{m}\ \ (\mathsf{t}_1\ \mathsf{x}_1, ..., \mathsf{t}_\mathsf{q}\ \mathsf{x}_\mathsf{q})\ \{\ \mathsf{c}_1{\Downarrow}\mathsf{c}'_1, ..., \mathsf{c}_\mathsf{n}{\Downarrow}\mathsf{c}'_\mathsf{n}\ \}\ \{\ \mathsf{e}\ \}$$

where t is the result type, $\mathsf{t}_1, ..., \mathsf{t}_\mathsf{q}$ are the types of the formal parameters $\mathsf{x}_1, ..., \mathsf{x}_\mathsf{q}$, and e is the body.[6] The effect consists of atomic effects $\mathsf{c}_1{\Downarrow}\mathsf{c}'_1$,..., $\mathsf{c}_\mathsf{n}{\Downarrow}\mathsf{c}'_\mathsf{n}$, with $\mathsf{n} \geq 0$. Each atomic effect $\mathsf{c}_\mathsf{i}{\Downarrow}\mathsf{c}'_\mathsf{i}$ means that any object of a subclass of c_i may be re-classified to any subclass of c'_i.

The operational semantics of $\mathcal{F}ickle_3$ (which ignores effects annotations) differs from that of $\mathcal{F}ickle_{\mathrm{II}}$ [6] only in the evaluation of re-classification expressions, id$\Downarrow$c. Here we briefly present the signature of the rewriting relation, $\leadsto$, and the semantics of re-classification expressions, id$\Downarrow$d. A detailed description of the other rules of the operational semantics is given in Section 4 of [6]. The signature of the rewriting relation $\leadsto$ is:

$$\leadsto\ \ :\ progr \longrightarrow e \times store\ \ \longrightarrow\ \ (\,val\ \cup\ dev\,)\ \times store$$

The operational semantics rewrites pairs of expressions and stores into pairs of values, exceptions, or errors, and stores in the context of a program P. The store maps `this` to an address, parameters to values, and addresses to objects. Values are addresses, or the source language values as in Section 3. Addresses may point to objects, but *not* to other addresses, primitive values, or `null`. Thus, in $\mathcal{F}ickle_3$,

[5] The syntax of $\mathcal{F}ickle_3$ differs from the syntax of $\mathcal{F}ickle_{\mathrm{II}}$ [6] only in the keywords **root** and **state** (that are not present in $\mathcal{F}ickle_3$) and in the effect annotations occurring in methods' signatures. Section 2 follows a slightly more liberal syntax, with abstract classes, abstract methods, and the implicit use of `this` to access fields and methods from the current class.

[6] Extending $\mathcal{F}ickle_3$ to allow methods to have local variables would be straightforward. The typing rules for local variables would be the same as for parameters.

$$
\begin{array}{lll}
\mathit{progr} & ::= & \mathit{class}^* \\
\mathit{class} & ::= & \mathbf{class}\ c\ \mathbf{extends}\ c\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
\mathit{field} & ::= & \mathit{type}\ f \\
\mathit{meth} & ::= & \mathit{type}\ m\ \ (\mathit{par}^*\ )\ \ \mathit{eff}\ \{\ e\ \} \\
\mathit{type} & ::= & \mathbf{bool}\ |\ \ c \\
\mathit{par} & ::= & \mathit{type}\ x \\
a & ::= & c{\Downarrow}c \\
\mathit{eff} & ::= & \{\ a^*\ \} \\
e & ::= & \mathbf{if}\ \ e\ \ \mathbf{then}\ \ \ e\ \ \mathbf{else}\ \ e\ |\ \ \mathit{var} := e\ |\ \ e\ \ ;\ \ e\ |\ \ \mathit{sVal}\ \ | \\
 & & \mathtt{this}\ \ |\ \ \mathit{var}\ \ |\ \ \ \mathbf{new}\ c\ |\ \ e.m(\ e^*)\ |\ \ \mathit{id}{\Downarrow}c \\
\mathit{var} & ::= & x\ |\ \ e.\ f \\
\mathit{sVal} & ::= & \mathtt{true}\ \ |\ \ \mathtt{false}\ \ |\ \ \mathtt{null} \\
\mathit{id} & ::= & \mathtt{this}\ |\ \ x
\end{array}
$$

with the following conventions

$c\ ::= \mathsf{c}\ \ \|\ \ \mathsf{c}'\ \ \|\ \ \mathsf{c_i}\ \ \|\ \ \mathsf{d}\ \ \|\ \ ...$	for class names
$f\ ::= \mathsf{f}\ \ \|\ \ \mathsf{f}'\ \ \|\ \ \mathsf{f_i}\ \ \|\ \ ...$	for field names
$m ::= \mathsf{m}\ \ \|\ \ \mathsf{m}'\ \ \|\ \ \mathsf{m_i}\ \ \|\ \ ...$	for method names
$x\ ::= \mathsf{x}\ \ \|\ \ \mathsf{x}'\ \ \|\ \ \mathsf{x_i}\ \ \|\ \ ...$	for parameter names

Fig. 2. Syntax of $\mathcal{F}\mathit{ickle}_3$

as in Java, pointers are implicit, and there are no pointers to pointers. As we will show, execution of well-typed expressions never produces an error, although it may throw a null pointer exception. Stores are denoted with σ, addresses with ι, exceptions and errors with dv.

$$
\begin{array}{lcl}
\mathit{store} & = & (\{\mathtt{this}\} \longrightarrow \mathit{addr}\,)\ \ \cup\ \ (\,x \longrightarrow \mathit{val}\,)\ \ \cup\ \ (\,\mathit{addr} \longrightarrow \mathit{object}\,) \\
\mathit{val} & = & \mathit{sVal} \cup \mathit{addr} \\
\mathit{dev} & = & \{\mathtt{nullPntrExc}, \mathtt{Err}\} \\
\mathit{object} & = & \{\ \ [[\mathsf{f}_1 : \mathsf{v}_1, ..., \mathsf{f_r} : \mathsf{v_r}]]^{\mathsf{c}}\ \ |\ \ \mathsf{f}_1, ..., \mathsf{f_r} \text{ are fields names,} \\
 & & \qquad\qquad\qquad\qquad\qquad \mathsf{v}_1, ..., \mathsf{v_r} \in \mathit{val},\ \text{and}\ \ \mathsf{c} \text{ is a class name}\ \ \}
\end{array}
$$

To evaluate a re-classification expressions, $\mathsf{id}{\Downarrow}\mathsf{d}$, 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 least common superclass of c and d, and initialize the other fields of d according to their types. For example, for store σ_1, with $\sigma_1(\mathsf{x}_1) = \iota$, and $\sigma_1(\iota) = [[\mathsf{brave} :\ \mathtt{true}\ , \mathsf{sword} :\ \iota']]^{\mathsf{Prince}}$, $\sigma_1(\iota') = [[...]]^{\mathsf{Weapon}}$, we have $\mathsf{x}_1{\Downarrow}\mathsf{Frog}, \sigma_1 \leadsto_{\mathsf{P_{pl}}} \iota, \sigma_2$ where $\sigma_2 = \sigma_1[\iota \mapsto [[\mathsf{brave} : \mathtt{true}\ , \mathsf{pouch} :\ \mathtt{null}]]^{\mathsf{Frog}}]$. *I.e.* we obtain an object of class Frog with unmodified field brave.

4 Typing

The following assertions, defined in Fig. 3 (where $\mathcal{C}(\mathsf{P},\mathsf{c})$, formally defined in Appendix A, returns the definition of class c in program P), describe classes, types, the subclassing relationship, and the widening relationship between types:

$$\frac{\text{The definitions in } \mathsf{P} \text{ are unique}}{\vdash\ \mathsf{P}\ \diamond_u} \qquad \frac{\vdash\ \mathsf{P}\ \diamond_u \text{ and the class hierarchy of } \mathsf{P} \text{ is acyclic}}{\vdash\ \mathsf{P}\ \diamond_h}$$

$$\frac{\vdash\ \mathsf{P}\ \diamond_u}{\mathsf{Object} \sqsubseteq_\mathsf{P} \mathsf{Object}} \qquad \frac{\begin{array}{l}\vdash\ \mathsf{P}\ \diamond_u\\ \mathsf{P} = ...\ \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}'\boldsymbol{\{}\ ...\ \boldsymbol{\}}...\end{array}}{\begin{array}{c}\mathsf{c} \sqsubseteq_\mathsf{P} \mathsf{c}\\ \mathsf{c} \sqsubseteq_\mathsf{P} \mathsf{c}'\end{array}} \qquad \frac{\begin{array}{l}\mathsf{c} \sqsubseteq_\mathsf{P} \mathsf{c}'\\ \mathsf{c}' \sqsubseteq_\mathsf{P} \mathsf{c}''\end{array}}{\mathsf{c} \sqsubseteq_\mathsf{P} \mathsf{c}''}$$

$$\frac{\begin{array}{l}\vdash\ \mathsf{P}\ \diamond_h\\ \mathcal{C}(\mathsf{P},\mathsf{c}) =\ \textbf{class}\ \mathsf{c}...\end{array}}{\begin{array}{c}\mathsf{P} \vdash \mathsf{c}\ \diamond_{ct}\\ \mathsf{P} \vdash \mathsf{c}\ \diamond_t\end{array}} \qquad \frac{}{\mathsf{P} \vdash \mathsf{bool}\ \diamond_t} \qquad \frac{}{\mathsf{bool} \leq_\mathsf{P} \mathsf{bool}} \qquad \frac{\mathsf{c} \sqsubseteq_\mathsf{P} \mathsf{c}'}{\mathsf{c} \leq_\mathsf{P} \mathsf{c}'}$$

$$\mathsf{t}_1 \sqcup_\mathsf{P} \mathsf{t}_2 = \begin{cases} \mathsf{t} & \text{if } \mathsf{t}_1 \leq_\mathsf{P} \mathsf{t} \quad \mathsf{t}_2 \leq_\mathsf{P} \mathsf{t} \quad \forall \mathsf{t}'.(\mathsf{t}_1 \leq_\mathsf{P} \mathsf{t}' \text{ and } \mathsf{t}_2 \leq_\mathsf{P} \mathsf{t}') \Rightarrow \mathsf{t} \leq_\mathsf{P} \mathsf{t}' \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

Fig. 3. Programs with unique definitions, well-formed inheritance hierarchy, subclasses, types, widening, and lub on types

$$\frac{\Gamma = \{\mathsf{x}_1 : \mathsf{t}_1, ..., \mathsf{x}_n : \mathsf{t}_n, \mathtt{this} : \mathsf{c}\}}{\Gamma(\mathsf{id}) = \begin{cases} \mathsf{t}_i & \text{if } \mathsf{id} = \mathsf{x}_i \\ \mathsf{c} & \text{if } \mathsf{id} = \mathtt{this} \\ \mathcal{U}df & \text{otherwise} \end{cases}} \qquad \Gamma[\mathsf{id} \mapsto \mathsf{t}](\mathsf{id}') = \begin{cases} \mathsf{t} & \text{if } \mathsf{id}' = \mathsf{id} \\ \Gamma(\mathsf{id}') & \text{otherwise} \end{cases}$$

Fig. 4. Environment lookup and update

$\mathsf{P} \vdash \mathsf{c}\ \diamond_{ct}$ means "c is a class in P",
$\mathsf{P} \vdash \mathsf{t}\ \diamond_t$ means "t is a type in P" (*i.e.* either a class or **bool**),
$\mathsf{c} \sqsubseteq_\mathsf{P} \mathsf{c}'$ means "class c is a subclass of c' in P", and
$\mathsf{t} \leq_\mathsf{P} \mathsf{t}'$ means "type t' widens type t in P" (*i.e.* t subclass of, or identical to, t').

Environments, Γ, map parameter names to types, and the receiver `this` to a class. They have the form $\{\mathsf{x}_1 : \mathsf{t}_1, ...\mathsf{x}_n : \mathsf{t}_n, \mathtt{this} : \mathsf{c}\}$. Lookup, $\Gamma(\mathsf{id})$, and update, $\Gamma[\mathsf{id} \mapsto \mathsf{t}]$, are defined in Fig. 4.

An atomic effect is a pair of classes, $\mathsf{c} \Downarrow \mathsf{c}'$, meaning that any object of a subclass of c may be re-classified to any subclass of c'. An effect, ϕ, is a set $\{\ \mathsf{c}_1 \Downarrow \mathsf{c}'_1, ..., \mathsf{c}_n \Downarrow \mathsf{c}'_n\ \}$ of atomic effects; it means that any object of a subclass of c_i may be re-classified to any subclass of c'_i. The empty effect, $\{\ \}$, guarantees that no object is re-classified. We say that the effect $\{\ \mathsf{c}_1 \Downarrow \mathsf{c}'_1, ..., \mathsf{c}_n \Downarrow \mathsf{c}'_n\ \}$ is well formed in P, and write $\mathsf{P} \vdash \{\ \mathsf{c}_1 \Downarrow \mathsf{c}'_1, ..., \mathsf{c}_n \Downarrow \mathsf{c}'_n\ \}\ \diamond$, to mean that

1. c_1,...,c_n are not subclasses of each other in P, and
2. for all $\mathsf{i}, \mathsf{j} \in \{1, ..., \mathsf{n}\}$, if ($\mathsf{c}'_i$ is a subclass of c_j) or (c_j is subclass of c'_i), then c'_j is a subclass of c'_i.

These two requirements simplify the definition of the operation of application of an effect to a type (that will be defined in Section 4.1). The first requirement assures that, for any class c, there is at most one atomic effect saying that objects belonging to class c might be re-classified. The second requirement assures that there is no atomic effect saying that objects belonging to a subclass of the target class c' of another atomic effect might be re-classified to a class that is not a subclass of c'. For instance, $\{\mathsf{Prince}\Downarrow\mathsf{Frog}, \mathsf{Frog}\Downarrow\mathsf{Prince}\}$ is not a well-formed effect, since it does not satisfy the second requirement. The effect $\{\mathsf{Prince}\Downarrow\mathsf{Player}, \mathsf{Frog}\Downarrow\mathsf{Player}\}$ is, instead, well-formed.

4.1 Typing Rules for Expessions

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

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma' \parallel \phi$$

where t conservatively estimates the type of e after its evaluation, the environment Γ' contains conservative estimations of the types of `this` and of the parameters after evaluation of e, and ϕ conservatively estimates the re-classification effect of the evaluation of e on objects.[7]

The typing rules for expressions are given in Fig. 5.[8] We use the lookup functions $\mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f})$ and $\mathcal{M}(\mathsf{P},\mathsf{c},\mathsf{m})$ which return, respectively, the definition of the field f and of the method m in the class c, going through through the class hierarchy, if necessary (see Appendix A). We follow the convention that rules can be applied only if the types in the conclusion are defined. This is useful in rules (*cond*) and (*id*).

Consider the rule (*cond*) for conditionals. The branches of the conditional, e_1 and e_2, are typed in the environment Γ_0, *i.e.* the environment updated by typing the first expression, e. Rule (*cond*) uses least upper bounds on types, environments, and effects to determine a conservative approximation of the type, of the resulting environment, and of the effect of the conditional expression. With $\mathsf{t} \sqcup_\mathsf{P} \mathsf{t}'$ we denote the *least upper bound of* t *and* t' with respect to $\leq_\mathsf{P}$, when it exists[9] (see Fig. 3). With $\Gamma \sqcup_\mathsf{P} \Gamma'$ we denote the *least upper bound operation on environments in* P, defined by:

$$\Gamma \sqcup_\mathsf{P} \Gamma' = \{\mathsf{id} : (\mathsf{t} \sqcup_\mathsf{P} \mathsf{t}') \mid \Gamma(\mathsf{id}) = \mathsf{t} \text{ and } \Gamma'(\mathsf{id}) = \mathsf{t}'\}.$$

The *subeffecting relation,* $\sqsubseteq_\mathsf{P}$*,* defined by:

$$\phi \sqsubseteq_\mathsf{P} \phi' \text{ iff , for all } \mathsf{c}\Downarrow\mathsf{c}' \in \phi, \text{ there exists } \mathsf{d}\Downarrow\mathsf{d}' \in \phi' \text{such that } \mathsf{c} \sqsubseteq_\mathsf{P} \mathsf{d} \text{ and } \mathsf{c}' \sqsubseteq_\mathsf{P} \mathsf{d}',$$

[7] In $\mathcal{F}ickle_{\text{II}}$ [6], where *effect annotations are set of root classes*, typing has the format $\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma' \parallel \{\mathsf{c}_1, ..., \mathsf{c}_\mathsf{n}\}$, where the $\mathcal{F}ickle_{\text{II}}$-style effect annotation $\{\mathsf{c}_1, ..., \mathsf{c}_\mathsf{n}\}$ is equivalent to the $\mathcal{F}ickle_3$-style effect annotation $\{\mathsf{c}_1\Downarrow\mathsf{c}_1, ..., \mathsf{c}_\mathsf{n}\Downarrow\mathsf{c}_\mathsf{n}\}$, meaning that any object of a subclass of the root class c_i may be re-classified to any subclass of the root class c_i.

[8] Besides the fact that the typing rules of $\mathcal{F}ickle_3$ use the least upper bound operator on effects ($\sqcup_\mathsf{P}$) instead of the set-theoretic union ($\cup$), the only difference between the typing rules for expressions of $\mathcal{F}ickle_3$ (in Fig. 5) and the typing rules for expressions of $\mathcal{F}ickle_{\text{II}}$ (in Fig. 6 of [6]) is in rule (*recl*).

[9] Note that for any class c the least upper bound $\mathsf{c} \sqcup_\mathsf{P} \textbf{bool}$ does not exist.

$$\frac{P,\Gamma \vdash e : \mathsf{bool} \parallel \Gamma_0 \parallel \phi \qquad P,\Gamma_0 \vdash e_1 : t_1 \parallel \Gamma_1 \parallel \phi_1 \qquad P,\Gamma_0 \vdash e_2 : t_2 \parallel \Gamma_2 \parallel \phi_2}{P,\Gamma \vdash \textbf{if } e \ \textbf{ then } \ e_1 \ \textbf{ else } e_2 : t_1 \sqcup_P t_2 \parallel \Gamma_1 \sqcup_P \Gamma_2 \parallel \phi \sqcup_P \phi_1 \sqcup_P \phi_2} \ (\mathit{cond})$$

$$\frac{\begin{array}{l} P,\Gamma \vdash e : c \parallel \Gamma_0 \parallel \phi \\ P,\Gamma_0 \vdash e' : t \parallel \Gamma' \parallel \phi' \\ \mathcal{F}(P, \phi' @_P c, f) = t' \qquad t \leq_P t' \end{array}}{P,\Gamma \vdash e.f{:=}e' : t \parallel \Gamma' \parallel \phi \sqcup_P \phi'} \ (\mathit{a\text{-}field}) \qquad \frac{\begin{array}{l} P,\Gamma \vdash e : t' \parallel \Gamma' \parallel \phi \\ \Gamma'(x) = t \qquad t' \leq_P t \end{array}}{P,\Gamma \vdash x{:=}e : t' \parallel \Gamma' \parallel \phi} \ (\mathit{a\text{-}var})$$

$$\frac{\begin{array}{l} P,\Gamma \vdash e : c \parallel \Gamma' \parallel \phi \\ \mathcal{F}(P, c, f) = t \end{array}}{P,\Gamma \vdash e.f : t \parallel \Gamma' \parallel \phi} \ (\mathit{field}) \qquad \frac{\begin{array}{l} P,\Gamma \vdash e : t \parallel \Gamma_0 \parallel \phi \\ P,\Gamma_0 \vdash e' : t' \parallel \Gamma' \parallel \phi' \end{array}}{P,\Gamma \vdash e;\, e' : t' \parallel \Gamma' \parallel \phi \sqcup_P \phi'} \ (\mathit{seq})$$

$$\frac{}{\begin{array}{l} P,\Gamma \vdash \texttt{true} \ : \mathsf{bool} \parallel \Gamma \parallel \{\ \} \\ P,\Gamma \vdash \texttt{false} \ : \mathsf{bool} \parallel \Gamma \parallel \{\ \} \end{array}} \ (\mathit{bool}) \qquad \frac{P \vdash c \ \diamond_{ct}}{P,\Gamma \vdash \texttt{null} : c \parallel \Gamma \parallel \{\ \}} \ (\mathit{null})$$

$$\frac{}{P,\Gamma \vdash id : \Gamma(id) \parallel \Gamma \parallel \{\ \}} \ (\mathit{id}) \qquad \frac{P \vdash c \ \diamond_{ct}}{P,\Gamma \vdash \textbf{new } c : c \parallel \Gamma \parallel \{\ \}} \ (\mathit{new})$$

$$\frac{\begin{array}{l} P,\Gamma \vdash e_0 : c \parallel \Gamma_0 \parallel \phi_0 \\ P,\Gamma_{i-1} \vdash e_i : t'_i \parallel \Gamma_i \parallel \phi_i \ \ (\forall i \in \{1,...,n\}) \\ \mathcal{M}(P, (\phi_1 \sqcup_P \cdots \sqcup_P \phi_n) @_P c, m) \ = t\ m(t_1\ x_1, ..., t_n\ x_n)\ \phi\ \{\ ...\ \} \\ (\phi_{i+1} \sqcup_P \cdots \sqcup_P \phi_n) @_P t'_i \leq_P t_i \ \ (\forall i \in \{1,...,n\}) \end{array}}{P,\Gamma \vdash e_0.m(e_1, ..., e_n) : t \parallel \phi @_P \Gamma_n \parallel \phi \sqcup_P \phi_0 \sqcup_P \cdots \sqcup_P \phi_n} \ (\mathit{meth})$$

$$\frac{P \vdash \Gamma(id) \ \diamond_{ct}}{P,\Gamma \vdash id{\Downarrow}c : c \parallel (\{\Gamma(id){\Downarrow}c\} @_P \Gamma)[id{\mapsto}c] \parallel \{\Gamma(id){\Downarrow}c\}} \ (\mathit{recl})$$

Fig. 5. Typing rules for expressions

formalizes the fact that the effect ϕ is a conservative approximation of the effect ϕ'. The effect $\phi \sqcup_P \phi'$, defined by:

$$\begin{array}{l} \phi \sqcup_P \phi' = \textbf{let} \\ \qquad \phi_0 = \{c{\Downarrow}(c' \sqcup_P (\sqcup_P \{d' \mid d{\Downarrow}d' \in \phi' \text{ and } d \sqsubseteq_P c\})) \ \mid \ c{\Downarrow}c' \in \phi\} \\ \qquad\qquad \cup \ \ \{d{\Downarrow}(d' \sqcup_P (\sqcup_P \{c' \mid c{\Downarrow}c' \in \phi \text{ and } c \sqsubseteq_P d\})) \ \mid \ d{\Downarrow}d' \in \phi'\} \\ \qquad \textbf{in} \\ \qquad \{c_0 \Downarrow (c'_0 \sqcup_P (\sqcup_P \{d'_0 \mid d_0{\Downarrow}d'_0 \in \phi_0 \text{ and } (c'_0 \sqsubseteq_P d_0 \text{ or } d_0 \sqsubseteq_P c'_0)\})) \ \mid \ c_0{\Downarrow}c'_0 \in \phi_0\} \end{array}$$

is the *least upper bound of the effects* ϕ *and* ϕ' with respect to $\sqsubseteq_P$. Note that the least upper bound of well-formed effects is always defined and it is a well-formed effect. The two branches of a conditional may cause different re-classifications for `this` and the parameters. So, after the evaluation we can only assert that `this` and the parameters belong to the least upper bound of their relative classes in Γ_1 and Γ_2. We can prove that for this rule $\Gamma_1 \sqcup_P \Gamma_2$ is defined. On the other hand, the least upper bound of the types of the branches, $\mathsf{t}_1 \sqcup_{\mathsf{P}} \mathsf{t}_2$, may not be defined, in which case the rule cannot be applied.

Consider now the typing of assignments, *i.e.* rules (*a-field*) and (*a-var*). Evaluation of the right hand side may modify the type of the left hand side. In particular, in (*a-var*) evaluation of e can modify the type of x. This is taken into account by looking up x in the environment Γ'. Also, in rule (*a-field*) evaluation of e' may modify the class of the object e. For this purpose, we define the application of effects to types:

$$\{\, \mathsf{c}_1 \Downarrow \mathsf{c}'_1, \ldots, \mathsf{c}_n \Downarrow \mathsf{c}'_n \,\} @_{\mathsf{P}} \mathsf{t} = \begin{cases} \mathsf{c}'_i \sqcup_{\mathsf{P}} \mathsf{t} & \text{if } \mathsf{t} \sqsubseteq_{\mathsf{P}} \mathsf{c}_i \text{ for some } \mathsf{i} \in 1, \ldots, \mathsf{n} \\ \mathsf{c}'_{i_1} \sqcup_{\mathsf{P}} \ldots \sqcup_{\mathsf{P}} \mathsf{c}'_{i_m} \sqcup_{\mathsf{P}} \mathsf{t} & \text{if } \mathsf{i}_j \in 1, \ldots, \mathsf{n} \ (\mathsf{j} \in 1, \ldots, \mathsf{m} \text{ and } \mathsf{m} \geq 1) \\ & \quad \text{are all the indexes such that } \mathsf{c}_{i_j} \sqsubseteq_{\mathsf{P}} \mathsf{t} \\ \mathsf{t} & \text{otherwise} \end{cases}$$

For example, $\{\, \mathsf{Frog} \Downarrow \mathsf{Prince} \,\} @_{\mathsf{P}_{pl}} \mathsf{Frog} = \mathsf{Player}$, $\{\, \mathsf{Frog} \Downarrow \mathsf{Prince} \,\} @_{\mathsf{P}_{pl}} \mathsf{Prince} = \mathsf{Prince}$, and $\{\, \mathsf{Frog} \Downarrow \mathsf{Prince} \,\} @_{\mathsf{P}_{pl}} \mathsf{Princess} = \mathsf{Princess}$.

In rule (*a-field*), by applying ϕ' to c before looking up f, we provide for the case where evaluation of e' might re-classify e and remove f in the process.

We say that *an effect ϕ respects a set of classes C in* P to mean that, for all $\mathsf{c} \in C$, $\phi @_{\mathsf{P}} \mathsf{c} = \mathsf{c}$.

Note that the field type cannot be changed since classes not respected by the effects listed in the program are not allowed to be used as types for fields. The effect of field assignment is the least upper bound of the effects of the left-hand side and of the right-hand side of the assignment, which conservatively approximates the application of the effect of the left-hand side followed by the application of the effect of the right-hand side. (This is due to the fact that: if $\mathsf{P} \vdash \phi \diamond$, $\mathsf{P} \vdash \phi' \diamond$, and $\mathsf{P} \vdash \mathsf{c} \diamond_{ct}$, then $\phi' @_{\mathsf{P}} (\phi @_{\mathsf{P}} \mathsf{c}) \sqsubseteq_{\mathsf{P}} (\phi \sqcup_{\mathsf{P}} \phi') @_{\mathsf{P}} \mathsf{c}$).

Consider now (*recl*): $\mathsf{id} \Downarrow \mathsf{c}$ is type correct if $\Gamma(\mathsf{id})$ (the type of id before the re-classification) is a class; note that, since the class hierarchy is a tree, this implies that c (the target of the re-classification) and $\Gamma(\mathsf{id})$ have a common superclass. The typing rule for re-classification updates the environment by changing the class of the identifier id. Moreover, since there could be aliasing with identifiers of classes that are subclasses or superclasses of the class of id, the static type of all such variables is set to the least upper bound of the current type and of the target of the re-classification. For this reason, we define the application of effects to environments:

$$\phi @_{\mathsf{P}} \Gamma = \{ \mathsf{id} : \ \phi @_{\mathsf{P}} \mathsf{t} \mid \Gamma(\mathsf{id}) = \mathsf{t} \}$$

For example, for an environment Γ_1, with $\Gamma_1(\mathsf{x}_1) = \Gamma_1(\mathsf{x}_2) = \mathsf{Frog}$, $\Gamma_1(\mathsf{x}_3) = \mathsf{Prince}$, we have $\{\, \mathsf{Frog} \Downarrow \mathsf{Prince} \,\} @_{\mathsf{P}_{pl}} \Gamma_1 = \Gamma_2$, where $\Gamma_2(\mathsf{x}_1) = \Gamma_2(\mathsf{x}_2) = \mathsf{Player}$ and $\Gamma_2(\mathsf{x}_3) = \mathsf{Prince}$. Therefore, the following typing judgement can be derived:

$$\mathsf{P}_{pl}, \Gamma_1 \ \vdash \ \mathsf{x}_2 \Downarrow \mathsf{Prince} \ : \ \mathsf{Prince} \parallel \Gamma_3 \parallel \ \{\, \mathsf{Frog} \Downarrow \mathsf{Prince} \,\}$$

where $\Gamma_3(\mathsf{x}_1) = \mathsf{Player}$, but $\Gamma_3(\mathsf{x}_2) = \Gamma_3(\mathsf{x}_3) = \mathsf{Prince}$.

Consider rule (*meth*) for method calls, $\mathsf{e}_0.\mathsf{m}(\mathsf{e}_1, \ldots, \mathsf{e}_n)$. The evaluation of the arguments $\mathsf{e}_{i+1}, \ldots, \mathsf{e}_n$ may modify the types of the arguments $\mathsf{e}_1, \ldots, \mathsf{e}_i$ and of the object e_0. This could happen if the original type of e_j $(0 \leq \mathsf{j} \leq \mathsf{i})$ is a subclass or superclass of the left-hand side of an atomic effect among the effects of $\mathsf{e}_{i+1}, \ldots, \mathsf{e}_n$. The definition of m has to be found in the new class of the object e_0, and the types of the formal parameters must be compared with the new types of $\mathsf{e}_1, ..., \mathsf{e}_{n-1}$. In (*meth*) we look up the definition of m in the class obtained by applying the effect of the arguments to the class of the receiver and we compare the types of formal and actual parameters by keeping into account the effects of the actual parameters.

4.2 Rules for Well-Formed Classes and Programs

The rules for checking that a program is well-formed are listed in Fig. 6.[10] With **Effect**(P) we denote the effect

$$\sqcup_{\mathsf{P}}\{\phi \mid \phi \text{ occurs in the signature of a method defined in a class of } \mathsf{P}\}$$

and with **ClassesFT**(P) we denote the set of classes

$$\{\mathsf{c} \mid \mathsf{c} \text{ is used as type for a field defined in a class of } \mathsf{P}\}.$$

A program is well formed (*i.e.* $\vdash \mathsf{P} \diamond$) if the inheritance hierarchy is well-formed (*i.e.* $\vdash \mathsf{P} \diamond_h$), all its classes c are well-formed (*i.e.* $\mathsf{P} \vdash \mathsf{c} \diamond$), and the effects respect the classes used as types for fields (*i.e.* **Effect**(P) respects **ClassesFT**(P)). Fields may not redefine fields from superclasses, and methods may redefine superclass methods only if they have the same name, arguments, and result type, and their effect is a subeffect of that of the overridden method.[11] Method bodies must be well formed, must return a value appropriate for the method signature, and their effect must be a subeffect of that in the signature. See Fig. 6, where $\mathcal{C}(\mathsf{P},\mathsf{c})$ returns the definition of class c in program P, and the lookup functions $\mathcal{FD}(\mathsf{P},\mathsf{c},\mathsf{f})$, $\mathcal{MD}(\mathsf{P},\mathsf{c},\mathsf{m})$ return, respectively, the definition of field f and method m in class c (the formal definitions are given in Appendix A).

4.3 Soundness

Figure 7 introduces agreement notions between programs, stores, and values.[12] The judgement $\mathsf{P}, \sigma \vdash \mathsf{v} \prec \mathsf{t}$ is instrumental to the definition of $\mathsf{P}, \sigma \vdash \mathsf{v} \lhd \mathsf{t}$: it avoids the use of coinduction. The judgement $\mathsf{P}, \phi \vdash \sigma \lhd \sigma'$ guarantees that the differences from σ to σ' are "small"; in particular, only objects of a subclass

[10] The only differences between the rules for well-formed classes and programs of $\mathcal{F}ickle_3$ (in Fig. 6) and those of $\mathcal{F}ickle_{\text{II}}$ (in Fig. 7 of [6]) are the following. In rule (*wfc*), the rule for $\mathcal{F}ickle_3$ uses a different notion of well-formed effect and uses the subeffecting relation instead of the set-theoretic inclusion. In rule (*wfp*), the rule for $\mathcal{F}ickle_3$ has the additional requirement that the effects must respect the classes used as types for fields.

[11] Thus, in contrast to Java and C++, $\mathcal{F}ickle_3$ does not allow field shadowing, nor method overloading. These features can be included into $\mathcal{F}ickle_3$ adopting the approach from [7]. However, this would complicate the presentation unnecessarily.

[12] The definitions in Fig. 7 differs from the analogous definition introduced in [6] for $\mathcal{F}ickle_{\text{II}}$ in the rule for the judgment $\mathsf{P}, \phi \vdash \sigma \lhd \sigma'$ (which is crucial to prove Theorem 1).

$$\frac{\begin{array}{l}\mathcal{C}(\mathsf{P},\mathsf{c}) = \ \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \ \mathsf{c}'\ \{...\} \\ \forall \mathsf{f}:\ \mathcal{FD}(\mathsf{P},\mathsf{c},\mathsf{f}) = \mathsf{t}_0 \quad \Longrightarrow \qquad \mathsf{P} \vdash \mathsf{t}_0 \diamond_t \quad \text{and} \quad \mathcal{F}(\mathsf{P},\mathsf{c}',\mathsf{f}) = \mathcal{U}df \\ \forall \mathsf{m}:\ \mathcal{MD}(\mathsf{P},\mathsf{c},\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, ..., \mathsf{t}_n\ \mathsf{x}_n)\ \phi\ \{\ \mathsf{e}\ \} \quad \Longrightarrow \\ \qquad \mathsf{P} \vdash \phi \diamond \\ \qquad \mathsf{P}, \{\mathsf{x}_1 : \mathsf{t}_1, \ldots, \mathsf{x}_n : \mathsf{t}_n, \mathtt{this} : \mathsf{c}\} \ \vdash \ \mathsf{e} \ : \ \mathsf{t}' \parallel \Gamma' \parallel \phi' \\ \qquad \mathsf{t}' \leq_{\mathsf{P}} \mathsf{t} \\ \qquad \phi' \sqsubseteq_{\mathsf{P}} \phi \\ \qquad \mathcal{M}(\mathsf{P},\mathsf{c}',\mathsf{m}) = \mathcal{U}df \ \text{ or } (\mathcal{M}(\mathsf{P},\mathsf{c}',\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1, ..., \mathsf{t}_n\ \mathsf{x}_n)\ \phi''\ \{\ ...\ \} \ \text{ and } \ \phi \sqsubseteq_{\mathsf{P}} \phi'')\end{array}}{\mathsf{P} \vdash \mathsf{c} \diamond} \ (wfc)$$

$$\frac{\vdash \ \mathsf{P} \diamond_h \qquad \forall \mathsf{c}:\ \mathcal{C}(\mathsf{P},\mathsf{c}) \neq \mathcal{U}df \quad \Longrightarrow \quad \mathsf{P} \vdash \mathsf{c} \diamond \qquad \textbf{Effect}(\mathsf{P}) \text{ respects } \textbf{ClassesFT}(\mathsf{P})}{\vdash \mathsf{P} \diamond} \ (wfp)$$

Fig. 6. Rules for well-formed classes and programs

of a class occurring in the left-hand side of an atomic effect in ϕ may be re-classified. The judgement $\mathsf{P},\sigma \vdash \mathsf{v} \lhd \mathsf{t}$, guarantees that value v conforms to type t. In particular, it requires that when v is an address it corresponds to an object of some class c subclass of t, that the object contains all fields required in the description of c, and that the fields contain values which conform to their type in c. The judgement $\mathsf{P}, \Gamma \vdash \sigma \diamond$ guarantees that all object fields contain values which conform to their types in the class of the objects, and that all parameters and the receiver are mapped to values which conform to their types in Γ.

The type system is sound in the sense that a converging well-typed expression returns a value which agrees with the expression's type, or `nullPntrExc`.

Theorem 1 (Type Soundness). *For a well-formed program* P*, environment* Γ*, and expression* e*, such that* $\mathsf{P}, \Gamma \ \vdash \ \mathsf{e} \ : \ \mathsf{t} \parallel \Gamma' \parallel \phi$ *if* $\mathsf{P}, \Gamma \vdash \sigma \diamond$*, and* e, σ *converges then*

– *either* $\mathsf{e}, \sigma \rightsquigarrow_{\mathsf{P}} \mathsf{v}, \sigma'$*,* $\quad \mathsf{P}, \sigma' \vdash \mathsf{v} \lhd \mathsf{t}$*,* $\quad \mathsf{P}, \Gamma' \vdash \sigma' \diamond$*,*
– *or* $\mathsf{e}, \sigma \rightsquigarrow_{\mathsf{P}}$ `nullPntrExc`$, \sigma'$.

5 Conclusions

In this paper we have proposed the language $\mathcal{F}ickle_3$ which improves $\mathcal{F}ickle_{\mathrm{II}}$ [6] by providing a more expressive type and effect system, and removing the need for the qualifiers **root** and **state** for classes. This makes possible uses of classes that were not anticipated at their definition time, so that a same class can be used for re-classifiable objects in certain contexts and for non re-classifiable objects in others.

Keeping the **root** and **state** qualifiers results in the language $\mathcal{F}ickle_{\mathrm{III}}$, an extension of $\mathcal{F}ickle_{\mathrm{II}}$, described in [4].

$$\frac{\mathsf{v} = \texttt{true} \;\text{ or }\; \mathsf{v} = \texttt{false}}{\mathsf{P}, \sigma \vdash \mathsf{v} \prec \mathbf{bool}} \;(\mathbf{bool} \prec) \qquad \frac{\mathsf{P} \vdash \mathsf{t} \diamond_{ct}}{\mathsf{P}, \sigma \vdash \texttt{null} \prec \mathsf{t}} \;(\texttt{null} \prec)$$

$$\frac{\sigma(\iota) = [\![...]\!]^{\mathsf{c}} \quad \mathsf{c} \leq_{\mathsf{P}} \mathsf{t}}{\mathsf{P}, \sigma \vdash \iota \prec \mathsf{t}} \;(\iota \prec)$$

$$\frac{\mathsf{P}, \sigma \vdash \mathsf{v} \prec \mathsf{t} \quad \mathsf{v} \in sVal}{\mathsf{P}, \sigma \vdash \mathsf{v} \lhd \mathsf{t}} \;(sVal \lhd) \qquad \frac{\begin{array}{l}\sigma(\iota) = [\![...]\!]^{\mathsf{c}} \quad \mathsf{P}, \sigma \vdash \iota \prec \mathsf{t} \\ \forall \mathsf{f} \in \mathcal{F}s(\mathsf{P}, \mathsf{c}) : \; \mathsf{P}, \sigma \vdash \sigma(\iota)(\mathsf{f}) \prec \mathcal{F}(\mathsf{P}, \mathsf{c}, \mathsf{f})\end{array}}{\mathsf{P}, \sigma \vdash \iota \lhd \mathsf{t}} \;(\iota \lhd)$$

$$\frac{\begin{array}{l}\sigma(\texttt{this}) = \sigma'(\texttt{this}) \\ \sigma(\iota) = [\![...]\!]^{\mathsf{c}} \quad \Longrightarrow \quad \sigma'(\iota) = [\![...]\!]^{\mathsf{c}'} \text{ and } \begin{cases} \mathsf{c}' \sqsubseteq_{\mathsf{P}} \mathsf{c} \sqcup_{\mathsf{P}} \mathsf{d}' & \text{if } \mathsf{c} \sqsubseteq_{\mathsf{P}} \mathsf{d} \text{ for some } \mathsf{d} \Downarrow \mathsf{d}' \in \phi \\ \mathsf{c}' = \mathsf{c} & \text{otherwise} \end{cases}\end{array}}{\mathsf{P}, \phi \vdash \sigma \lhd \sigma'} \;(\sigma \lhd)$$

$$\frac{\begin{array}{l}\sigma(\iota) = [\![...]\!]^{\mathsf{c}} \quad \Longrightarrow \quad \mathsf{P}, \sigma \vdash \iota \lhd \mathsf{c} \quad \text{(for all addresses } \iota) \\ \Gamma(\mathsf{id}) \neq \mathcal{U}df \quad \Longrightarrow \quad \mathsf{P}, \sigma \vdash \sigma(\mathsf{id}) \lhd \Gamma(\mathsf{id}) \quad \text{(for all identifiers } \mathsf{id})\end{array}}{\mathsf{P}, \Gamma \vdash \sigma \diamond} \;(\diamond)$$

Fig. 7. Agreement between programs, stores, and values

More experimentation with $\mathcal{F}ickle_3$ and $\mathcal{F}ickle_{\mathrm{III}}$ is needed in order to assess the "real usefulness" of their features and to compare the two languages from the "software engineering point of view".

The paper [1] provides a translation of $\mathcal{F}ickle$ into Java. The same translation scheme could be used for $\mathcal{F}ickle_3$ and $\mathcal{F}ickle_{\mathrm{III}}$.

Acknowledgements

We would like to thank Mariangiola Dezani and the anonymous ICTCS'03 referees for useful comments and suggestions.

References

1. D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, and E. Zucca. An Effective Translation of Fickle into Java. In *ICTCS'01*, volume 2002 of *LNCS*, pages 215–234, Berlin, 2001. Springer.
2. C. Chambers. Predicate Classes. In *ECOOP'93*, volume 707 of *LNCS*, pages 268–296, Berlin, 1993. Springer.
3. P. Costanza. Dynamic Object Replacement and Implementation-Only Classes. In *WCOP'01 (at ECOOP'01)*, 2001. Available from http://www.cs.uni-bonn.de/∼costanza/implementationonly.pdf.

4. F. Damiani, M. Dezani-Ciancaglini, S. Drossopoulou, and P. Giannini. Refined Effects for Re-classification: Fickle$_{III}$. Report for the IST-2001-33477 DART project - available at the url http://www.cee.hw.ac.uk/DART/reports/D3.1/DDDG02b.pdf, 2002.
5. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic Object Re-classification. In *ECOOP'01*, volume 2072 of *LNCS*, pages 130–149, Berlin, 2001. Springer. A shorter version is available in: Electronic proceedings of FOOL8 (http://www.cs.williams.edu/∼kim/FOOL/).
6. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More Dynamic Object Re-classification: Fickle$_{II}$. *ACM Transactions On Programming Languages and Systems*, 24(2):153–191, 2002.
7. Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.
8. G. Ghelli and D. Palmerini. Foundations of Extended Objets with Roles (*extended abstract*). In *FOOL6*, 1999. Available from http://www.cs.williams.edu/∼kim/FOOL/FOOL6.html.
9. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *POPL'88*, pages 47–57, New York, 1988. ACM Press.
10. M. Serrano. Wide Classes. In *ECOOP'99*, volume 1628 of *LNCS*, pages 391–415, Berlin, 1999. Springer.
11. A. Tailvasaari. Object Oriented Programming with Modes. *Journal of Object Oriented Programming*, 6(3):27–32, 1993.
12. J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

A Definitions Concerning Lookup

For program P with $\vdash\ \mathsf{P}\ \diamond_u$ (see Fig. 3) and identifier $\mathsf{c}\neq\mathsf{Object}$, we define the lookup of the class declaration for c:

$$\mathcal{C}(\mathsf{P},\mathsf{c}) = \begin{cases} \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}'\{\mathsf{cBody}\} & \textit{if}\ \mathsf{P}=\mathsf{P}'\ \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}'\{\mathsf{cBody}\}\ \mathsf{P}'', \\ \mathcal{U}\!\mathit{df} & \textit{otherwise} \end{cases}$$

For program P with $\vdash\ \mathsf{P}\ \diamond_h$ (see Fig. 3), identifier c such that

$$\mathcal{C}(\mathsf{P},\mathsf{c}) = \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}'\{\mathsf{cBody}\},$$

and identifiers f and m we define:

$$\mathcal{FD}(\mathsf{P},\mathsf{c},\mathsf{f}) = \begin{cases} \mathsf{t} & \textit{if}\ \mathsf{cBody} = \ldots\ \mathsf{t}\ \mathsf{f}\ \ldots \\ \mathcal{U}\!\mathit{df} & \textit{otherwise} \end{cases}$$

$$\mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f}) = \begin{cases} \mathcal{FD}(\mathsf{P},\mathsf{c},\mathsf{f}) & \textit{if}\ \ \mathcal{FD}(\mathsf{P},\mathsf{c},\mathsf{f}) \neq \mathcal{U}\!\mathit{df}, \\ \mathcal{F}(\mathsf{P},\mathsf{c}',\mathsf{f}) & \textit{otherwise} \end{cases}$$

$$\mathcal{F}(\mathsf{P},\mathsf{Object},\mathsf{f}) = \mathcal{U}\!\mathit{df}$$

$$\mathcal{F}s(\mathsf{P},\mathsf{c}) = \{\mathsf{f} \mid \mathcal{F}(\mathsf{P},\mathsf{c},\mathsf{f}) \neq \mathcal{U}\!\mathit{df}\}$$

$$\mathcal{MD}(\mathsf{P},\mathsf{c},\mathsf{m}) = \begin{cases} \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x}_1,\ldots,\mathsf{t}_n\ \mathsf{x}_n)\,\phi\,\{\ \mathsf{e}\ \} & \textit{if}\ \mathsf{cBody} = \ldots\mathsf{t}\ \mathsf{m}\ (\mathsf{t}_1\ \mathsf{x}_1\ldots\mathsf{t}_n\ \mathsf{x}_n)\phi\{\mathsf{e}\}\ldots \\ \mathcal{U}\!\mathit{df} & \textit{otherwise} \end{cases}$$

$$\mathcal{M}(\mathsf{P},\mathsf{c},\mathsf{m}) = \begin{cases} \mathcal{MD}(\mathsf{P},\mathsf{c},\mathsf{m}) & \textit{if}\ \ \mathcal{MD}(\mathsf{P},\mathsf{c},\mathsf{m}) \neq \mathcal{U}\!\mathit{df}, \\ \mathcal{M}(\mathsf{P},\mathsf{c}',\mathsf{m}) & \textit{otherwise} \end{cases}$$

$$\mathcal{M}(\mathsf{P},\mathsf{Object},\mathsf{m}) = \mathcal{U}\!\mathit{df}$$