Re-classification and Multi-threading: $\mathcal{F}ickle_{\mathtt{MT}}$ [*]

[Extended Abstract]

Ferruccio Damiani[†]
Dipartimento di Informatica,
Università di Torino
Corso Svizzera 185
10149 Torino, Italy
damiani@di.unito.it

Mariangiola Dezani-Ciancaglini[‡]
Dipartimento di Informatica,
Università di Torino
Corso Svizzera 185
10149 Torino, Italy
dezani@di.unito.it

Paola Giannini[§]
Dipartimento di Informatica,
Università del Piemonte Orientale
Corso Borsalino 54
15100 Alessandria, Italy
giannini@mfn.unipmn.it

ABSTRACT

In this paper we consider re-classification in the presence of multi-threading. To this aim we define a multi-threaded extension of the language $\mathcal{F}ickle$, that we call $\mathcal{F}ickle_{\mathtt{MT}}$. We define an operational semantics and a type and effect system for the language. Each method signature carries the information on the possible effects of the method execution. The type and effect system statically checks this information. The operational semantics uses this information in order to delay the execution of some threads when this could cause messageNotUnderstood errors. We prove that in the execution of a well-typed expression such delays do not produce deadlock.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects inheritance polymorphism*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*object-oriented constructs type structure*

[†]Partially supported by EU within the FET - Global Computing initiative, project DART IST-2001-33477 and by MURST Cofin'01 project NAPOLI.

[‡]Partially supported by EU within the FET - Global Computing initiative, project MIKADO IST-2001-32222 and by MURST Cofin'02 project McTati.

[§]Partially supported by EU within the FET - Global Computing initiative, project DART IST-2001-33477, by MURST Cofin'01 project NAPOLI, and by MURST Cofin'02 project McTati.

[*]The funding bodies are not responsible for any use that might be made of the results presented here.

General Terms

Theory, Languages

Keywords

Object-oriented Languages, Type and Effect Systems, Multi-threading

1. INTRODUCTION

Re-classifiable objects support the changing of object's behaviour by changing class membership at runtime, see e.g. [3, 11, 9, 13, 6, 7]. The language $\mathcal{F}ickle$ [6, 7] is particularly interesting since it is a Java-like language which combines features for object re-classification with a strong type system.

In this paper we consider the issue of dealing with re-classification in the presence of multi-threading. To this aim we define a multi-threaded extension of the language $\mathcal{F}ickle$, that we call $\mathcal{F}ickle_{\mathtt{MT}}$. In $\mathcal{F}ickle_{\mathtt{MT}}$ the expression **spawn**(e) starts the evaluation of the expression e in a new thread while the current thread continues by evaluating the expression following **spawn**(e). The new and the current thread work on a common heap containing the set of defined objects, and, through aliasing, re-classification may change the class membership of objects across threads.

The basic problem in the design of languages with re-classification features is to ensure that, even though objects may be re-classified across classes with different members, no attempt is made to access non-existing members of an object. In the single-threaded language $\mathcal{F}ickle$ [6, 7, 4] this is achieved by the use of a *static type and effect system* that conservatively estimates the re-classifications that may be caused by the execution of expressions and changes the types of the variables that may refer to a re-classified object. In a multi-threaded environment this is not enough, since the object referred to by a given variable could be re-classified by another thread. Therefore, we have to prevent executions in which a thread re-classifies an object while another is executing a method on the object. We achieve this by combining a static type and effect system with a *synchronization mechanism* based on *effect information.*

Each method declaration gathers in addition to information on the classes of the objects that may be re-classified (as in $\mathcal{F}ickle$) also information on the classes of the objects that may receive messages. The operational semantics uses the

$$
\begin{array}{lcl}
P & ::= & \mathit{class}^* \\
\mathit{class} & ::= & [\,\mathbf{root} \mid \mathbf{state}\,]\ \mathbf{class}\ \mathsf{c}\ \ \mathbf{extends}\ \mathsf{c}\ \{\mathit{field}^*\ \ \mathit{meth}^*\} \\
\mathit{field} & ::= & \mathsf{t}\ \mathsf{f} \\
\mathit{meth} & ::= & \mathsf{t}\ \mathsf{m}\ (\mathsf{t}\ \mathsf{x})\ \Theta\ \ \{\,\mathsf{e}\,\} \\
\mathsf{t} & ::= & \mathbf{bool} \mid\ \mathsf{c} \\
\Theta & ::= & \langle\{\mathsf{c}^*\},\{\mathsf{c}^*\}\rangle \\
\mathsf{e} & ::= & \mathbf{if}\ \mathsf{e}\ \mathbf{then}\ \mathsf{e}\ \mathbf{else}\ \mathsf{e} \mid\ \mathsf{e}\,;\mathsf{e} \mid\ \ \mathbf{new}\,\mathsf{c} \mid\ \mathsf{v} \\
 & \mid & \mathtt{this}\ \mid\ \mathsf{x}{:=}\mathsf{e} \mid\ \mathsf{e.f}{:=}\mathsf{e} \mid\ \mathsf{x} \mid\ \mathsf{e.f} \mid\ \mathsf{e.m}\ (\,\mathsf{e}\,) \\
 & \mid & \mathtt{this}{\Downarrow}\mathsf{c} \mid \mathbf{spawn}(\mathsf{e}) \\
\mathsf{v} & ::= & \mathtt{true}\ \mid\ \mathtt{false}\ \mid\ \mathtt{null}
\end{array}
$$

with the following conventions

$\mathsf{c}, \mathsf{c}', \mathsf{c}_i, \mathsf{d} \ldots$	for class names
$\mathsf{f}, \mathsf{f}', \mathsf{f}_i \ldots$	for field names
$\mathsf{m}, \mathsf{m}', \mathsf{m}_i \ldots$	for method names

Figure 1: Syntax of $\mathcal{F}ickle_{\mathtt{MT}}$

previous information to delay threads that either re-classify objects that are used by other threads, or invoke methods on objects that could be re-classified. We can prove that:

- no execution of a well-typed expression can cause the access to non-existing members of objects, and
- the delays introduced do not cause deadlocks.

We model multi-threading at a rather abstract level, since our aim is to study its interaction with re-classification. Other work on multi-threaded Java-like languages (*e.g.* [2, 10]), instead, consider a semantics closer to the implementation. Moreover, we do not consider explicit synchronisation constructs (like the "synchronised method", "wait", and "notify" constructs that can be found in Java) since they are orthogonal to re-classification.

This paper is organised as follows: Section 2 provides a brief overview of $\mathcal{F}ickle_{\mathtt{MT}}$. Section 3 presents the operational semantics and Section 4 introduces the type and effect system. The main results of the paper, namely the soundness and progress theorems, are stated in Section 5. A version of the paper with proofs is available at the url http://www.di.unito.it/∼damiani/papers/fickleMT.html.

2. $\mathcal{F}ickle_{\mathtt{MT}}$ IN A NUTSHELL

$\mathcal{F}ickle_{\mathtt{MT}}$ is a typed, imperative, class-based language, where classes are types, subclasses are subtypes, and methods are defined inside classes and selected depending on the class of the object on which the method is invoked. The syntax of $\mathcal{F}ickle_{\mathtt{MT}}$ is given by the pseudo-grammar in Figure 1, where a [-] pair means optional, and A^* means zero or more repetitions of A. We omit separators like "**;** " or "**,**" where they are obvious. Programs are ranged over by P (with subscripts and superscript when needed), types by t, effects by Θ, expressions by e, and values by v.

A program is a sequence of class definitions. A class definition may be preceded by the keyword **root** or **state**. All the subclasses of a *root* class must be *state* classes, and all the superclasses of a root class must be non-root/non-state classes. Root and state classes are the possible sources and targets of re-classification - static typing guarantees that the source and the target of a re-classification are subclasses of a same root class.

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

The expression $\mathtt{this}{\Downarrow}\mathsf{c}$ changes the class c' of the object pointed at by `this` to c, the values of all the fields declared in the common root superclass of c' and c are preserved, and the other fields of c are initialized to the default value associated to their type (`true` for **bool** and `null` for classes).

Field's types, method's result types and parameter's types cannot be state classes - so that, even though objects pointed at by them may be re-classified across classes with different members, there will never be an attempt to access non-existing members. In contrast, the static type of `this` may be a state class. To ensure that no attempt to access non-existing members will take place, the (possible) re-classification of the object pointed at by `this` will affect the static type of `this`.

Methods have only one parameter and are declared by

$$\mathsf{t}\,\mathsf{m}\ (\mathsf{t}'\,\mathsf{x})\ \Theta\ \ \{\,\mathsf{e}\,\}$$

where t is the result type, t' is the type of the formal parameter x, Θ is the effect, and e is the method's body. The effect Θ is a pair $\langle\phi, \psi\rangle$ where

ϕ, called *re-classification effect*, is a set of root classes which conservatively estimates the set of classes whose objects could be re-classified during the evaluation of e, and

ψ, called *receive effect*, is a set of classes which conservatively estimates the set of classes whose objects could receive a method call during the evaluation of e.

The expression **spawn**(e) causes the execution of the expression e in a new thread. The receive effect ψ, used only by the synchronization mechanism, plays a crucial role in guaranteeing safety in presence of multi-threading.[1]

Typing the body of a method involves the use of *environments*, Γ, mapping the parameter name to a type, and the metavariable `this` to a class. Environments are denoted by $\{\mathsf{t}\ \mathsf{x}, \mathsf{c}\ \mathtt{this}\}$; lookup, $\Gamma(\mathsf{id})$, and update, $\Gamma[\mathsf{id}{\mapsto}\mathsf{t}]$, have the usual meaning.

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

$$P, \Gamma\ \vdash\ \mathsf{e}\ :\ \mathsf{t}\ [\![\ \mathsf{c}\ ]\!]\ \Theta$$

where t is the type of the value returned by evaluation of e, c is the type of `this` after execution of e, Θ is the effect of e.

2.1 An Example

In Fig. 2 we give the program P_{pl}, which defines some classes inspired to adventure games.[2]

In the typing judgements of $\mathcal{F}ickle_{\mathtt{MT}}$, as in [6], the class c and the *re-classification effect* ϕ are used to track how the receivers of methods can change class. For example, if $\Gamma(\mathtt{this})$=Frog and $\Gamma'(\mathtt{this})$=Prince we have:

$$
\begin{array}{l}
P_{\mathsf{pl}}, \Gamma\ \vdash\ \mathtt{this.pouch{:=}\ null}\ :\ \mathsf{Vocal}\ [\![\ \mathsf{Frog}\ ]\!]\ \langle\{\},\{\}\rangle \\
P_{\mathsf{pl}}, \Gamma\ \vdash\ \mathtt{this}{\Downarrow}\mathsf{Prince}\ :\ \mathsf{Prince}\ [\![\ \mathsf{Prince}\ ]\!]\ \langle\{\mathsf{Player}\},\{\}\rangle \\
P_{\mathsf{pl}}, \Gamma'\ \vdash\ \mathtt{this.sword{:=}}\ \mathbf{new}\ \mathsf{Weapon}\ :\ \mathsf{Weapon}\ [\![\ \mathsf{Prince}\ ]\!]\ \langle\{\},\{\}\rangle
\end{array}
$$

[1]In [6, 7, 4], where the evaluation is sequential, the component ψ is missing.

[2]This example is a multi-threaded variant of an example proposed in [7].

```
class Weapon extends Object{bool swing()⟨{ }, { }⟩{···}}

class Vocal extends Object{bool blow()⟨{ }, { }⟩{···}}

abstract root class Player extends Object{
   bool brave;
   abstract bool wake() ⟨{ }, {Weapon, Vocal}⟩;
   abstract Weapon kissed() ⟨ { Player } , { }⟩;
   abstract Vocal cursed()⟨ { Player } , { }⟩;
}

state class Frog extends Player{
   Vocal pouch;
   bool wake()⟨{ }, {Vocal}⟩{this.pouch.blow() ; this.brave}
   Weapon kissed()⟨ { Player } , { }⟩
      {this.pouch:= null;
       this⇓Prince;
       this.sword:= new Weapon}
   Vocal cursed()⟨{ }, { }⟩{this.pouch}
}

state class Prince extends Player{
   Weapon sword;
   bool wake()⟨{ }, {Weapon}⟩{this.sword.swing(); this.brave}
   Weapon kissed()⟨{ }, { }⟩{this.sword}
   Vocal cursed()⟨ { Player } , { }⟩
      {this⇓Frog; this.pouch:= new Vocal}
}

class Game extends Object{
   bool play(Player x)⟨{ }, { }⟩
      {spawn(x.wake();  x.kissed());              /* row (1)*/
       spawn(x.wake();  x.wake())}                /* row (2)*/
}
```

Figure 2: Program P_{pl}- players with re-classifications

so the method kissed in class Frog is well typed. The *receiver effect* ψ has been added to correctly deal with multi-threading. Consider for instance the expression:

$$(\mathbf{new}\ \mathsf{Game}).\mathsf{play}(\mathbf{new}\ \mathsf{Frog})$$

The method play spawns two threads. We need to avoid an execution in which the receiver of the method wake in the thread created in row (2), which is initially a Frog, becomes (due to the concurrent execution method kissed in the thread created in row (1)) a Prince before it "blew its pouch" (according to the body of wake in class Frog), since this would produce a messageNotUnderstood error. This is realized taking into account that, for Γ(this)=Game and Γ(x)=Player, the two method calls have typing

$$P_{\mathsf{pl}}, \Gamma \ \vdash\ \mathsf{x.kissed()}\ :\ \mathsf{Weapon} \ \| \ \mathsf{Player} \ \| \ \langle\{\mathsf{Player}\}, \{\mathsf{Player}\}\rangle$$

and typing

$$P_{\mathsf{pl}}, \Gamma \ \vdash\ \mathsf{x.wake()}\ :\ \mathbf{bool} \ \| \ \mathsf{Player} \ \| \ \langle\{\}, \{\mathsf{Player}, \mathsf{Vocal}, \mathsf{Weapon}\}\rangle$$

respectively. The crucial observation is that there are objects in the heap (like the object pointed by x) which could be re-classified by the method kissed in one thread and could at the same time be used as receiver of the method wake in another thread. This is made explicit by the fact that the class Frog of the object pointed at by x has a super-class (i.e. Player) both in the re-classification effect of x.kissed () and in the receiver effect of x.wake ().

To avoid these problems, we require that each object that could be re-classified by one thread is not re-classified or used as receiver by other threads. This is realised by introducing mappings from addresses to integers, a global one (the mapping γ) and a local one for each thread (the mappings λ). These mappings identify the objects that may be re-classified and the objects that may receive messages by the active methods. Before executing a method call the rules of the operational semantics check (using the information given by the method effect) these conditions and delay the execution of the method call until they are satisfied.

For instance, in the expression (**new** Game).play(**new** Frog) considered above, the method call x.kissed() (in the thread created) in row (1) must be executed either before the execution of the first method call x.wake() in row (2), or between the two method calls x.wake() in row (2) (that is when the first call is terminated and the second is not started), or after the execution of the second call x.wake() in row (2). Note that the simultaneous execution of method calls x.wake() in different threads is allowed, since it cannot cause a messageNotUnderstood error.

The creation of new objects modifies the mappings γ and λ to ensure that all inner calls can be executed, guaranteeing the absence of deadlocks.

3. OPERATIONAL SEMANTICS

The semantics of $\mathcal{F}ickle_{\mathtt{MT}}$ is a small step operational semantics, presented in the style advocated in [14]. To model multi-threaded computations we consider *multi-threaded configurations* composed by exactly one heap, one "global objects state" (which are shared by all the threads) and one set of tuples of *single-threaded configurations* ⟨frame, "local objects state", effects, expression⟩ (one tuple for each thread). The "global objects state" and the "local objects state" keep the information about "which threads are active on which objects".

The evaluation of the expression **spawn**(e) in one of the existing threads creates a new thread that runs the expression e in parallel with the existing threads.

The semantics, which specifies how a multi-threaded configuration rewrites with respect to a program P, is defined by a reduction relation:

$$\longmapsto_P \subseteq (\mathbf{H} \times \mathbf{GOS} \times \mathcal{P}_{fin}(\mathbf{F} \times \mathbf{LOS} \times \mathbf{EF} \times \mathbf{E})) \ \times \\ (\mathbf{H} \times \mathbf{GOS} \times \mathcal{P}_{fin}(\mathbf{F} \times \mathbf{LOS} \times \mathbf{EF} \times \mathbf{E})) \cup \{\mathtt{exc}, \mathtt{end}\})$$

which is defined in terms of another reduction relation specifying how sequential configurations are rewritten:

$$\longrightarrow_P \subseteq (\mathbf{H} \times \mathbf{GOS} \times (\mathbf{F} \times \mathbf{LOS} \times \mathbf{EF} \times \mathbf{E})) \ \times \\ (\mathbf{H} \times \mathbf{GOS} \times (\mathbf{F} \times \mathbf{LOS} \times \mathbf{EF} \times \mathbf{E})) \cup \{\mathtt{exc}\})$$

A well-typed program terminates either normally with the special term end or with the special term exc modelling null pointer exceptions - which are the only source of abnormal termination in well typed programs.

Figure 3 gives the more interesting reduction rules for sequential reductions and Figure 4 all the rules for multi-threaded reductions.

The semantic categories involved in the definition of $\longmapsto_P$ are:

- *Addresses*, $\iota \in \mathbf{I}$ (we assume a denumerable set of addresses).

(call$^\mathsf{o}$) $\ll \chi, \gamma, \langle \rho, \lambda, \langle \{\,\}, \{\,\} \rangle, \iota.\mathsf{m}(\mathsf{v}) \rangle \gg \longrightarrow_P \ll \chi, \gamma', \langle \rho, \lambda', \Theta, \mathbf{return}^\mathsf{o}([\mathsf{x} \mapsto \mathsf{v}, \mathtt{this} \mapsto \iota], \mathsf{e}) \rangle \gg$

if $\forall \iota' \in \Phi(\gamma(\iota') = 0),\ \forall \iota' \in \Psi(\gamma(\iota') \geq 0),\ \ \chi(\iota) = [[\ldots]]^\mathsf{c}$ and $\mathcal{M}(P, \mathsf{c}, \mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x})\ \langle \phi, \psi \rangle\ \{\ \mathsf{e}\ \}$,

where $\Phi = \phi \star_P \chi,\ \Psi = \psi \star_P \chi \cup \{\iota\} - \Phi,\ \Theta = \langle \phi, \psi \cup \{\mathsf{c}\} \rangle$,
$\gamma' = \gamma[\iota' : -1 \mid \iota' \in \Phi][\iota' : \gamma(\iota') + 1 \mid \iota' \in \Psi]$, and $\lambda' = \lambda[\iota' : -1 \mid \iota' \in \Phi][\iota' : 1 \mid \iota' \in \Psi]$

(call$^\mathsf{i}$) $\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \iota.\mathsf{m}(\mathsf{v}) \rangle \gg \longrightarrow_P \ll \chi, \gamma', \langle \rho, \lambda', \Theta, \mathbf{return}^\mathsf{i}([\mathsf{x} \mapsto \mathsf{v}, \mathtt{this} \mapsto \iota], \mathsf{e}) \rangle \gg$

if $\Theta \neq \langle \{\,\}, \{\,\} \rangle$

(ret$^\mathsf{o}$) $\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \mathbf{return}^\mathsf{o}(\rho', \mathsf{v}) \rangle \gg \longrightarrow_P \ll \chi, \gamma', \langle \rho, \lambda_0, \langle \{\,\}, \{\,\} \rangle, \mathsf{v} \rangle \gg$

where $\gamma' = \gamma[\iota : 0 \mid \lambda(\iota) = -1][\iota : \gamma(\iota) - 1 \mid \lambda(\iota) = 1]$

(ret$^\mathsf{i}$) $\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \mathbf{return}^\mathsf{i}(\rho', \mathsf{v}) \rangle \gg \longrightarrow_P \ll \chi, \gamma, \langle \rho, \lambda, \Theta, \mathsf{v} \rangle \gg$

(new) $\ll \chi, \gamma, \langle \rho, \lambda, \langle \phi, \psi \rangle,\ \mathbf{new}\ \mathsf{c} \rangle \gg \longrightarrow_P \ll \chi', \gamma[\iota : \mathsf{b}], \langle \rho, \lambda[\iota : \mathsf{b}], \langle \phi, \psi \rangle, \iota \rangle \gg$

if $\chi(\iota)$ is undefined, $\mathcal{F}s(P, \mathsf{c}) = \{\mathsf{f}_1, \ldots, \mathsf{f}_\mathsf{r}\}$, and $\forall \mathsf{l} \in 1, \ldots, \mathsf{r} : \mathsf{v}_\mathsf{l}$ initial for $\mathcal{F}(P, \mathsf{c}, \mathsf{f}_\mathsf{l})$

where $\chi' = \chi[\iota \mapsto [[\mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_\mathsf{r} : \mathsf{v}_\mathsf{r}]]^\mathsf{c}]$

$$\mathsf{b} = \begin{cases} -1 & \text{if } \iota \in \phi \star_P \chi', \\ 1 & \text{if } \iota \in \psi \star_P \chi' \text{ and } \iota \notin \phi \star_P \chi', \\ 0 & \text{otherwise.} \end{cases}$$

(rec) $\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \mathtt{this} \Downarrow \mathsf{c} \rangle \gg \longrightarrow_P \ll \chi[\iota \mapsto [[\mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_{\mathsf{r}+\mathsf{q}} : \mathsf{v}_{\mathsf{r}+\mathsf{q}}]]^\mathsf{c}], \gamma, \langle \rho, \lambda, \Theta, \iota \rangle \gg$

where $\iota = \rho(\mathtt{this}), \chi(\iota) = [[\ldots]]^{\mathsf{c}'}$, $\mathcal{F}s(P, \mathcal{R}(\mathsf{P}, \mathsf{c})) = \{\mathsf{f}_1, \ldots, \mathsf{f}_\mathsf{r}\}$, $\forall \mathsf{l} \in 1, \ldots, \mathsf{r} : \mathsf{v}_\mathsf{l} = \chi(\iota)(\mathsf{f}_\mathsf{l})$,
$\mathcal{F}s(P, \mathsf{c}) \setminus \{\mathsf{f}_1, \ldots, \mathsf{f}_\mathsf{r}\} = \{\mathsf{f}_{\mathsf{r}+1}, \ldots, \mathsf{f}_{\mathsf{r}+\mathsf{q}}\}$, and $\forall \mathsf{l} \in \mathsf{r} + 1, \ldots, \mathsf{r} + \mathsf{q} : \mathsf{v}_\mathsf{l}$ initial for $\mathcal{F}(P, \mathsf{c}, \mathsf{f}_\mathsf{l})$

($:=_\mathsf{v}$) $\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \mathsf{x} := \mathsf{v} \rangle \gg \longrightarrow_P \ll \chi, \gamma, \langle \rho[\mathsf{x} : \mathsf{v}], \lambda, \Theta, \mathsf{v} \rangle \gg$

($:=_\mathsf{f}$) $\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \iota.f := \mathsf{v} \rangle \gg \longrightarrow_P \ll \chi[\iota : \chi(\iota)[f : \mathsf{v}]], \gamma, \langle \rho, \lambda, \Theta, \mathsf{v} \rangle \gg$, if $\chi(\iota)(f) \neq \mathcal{U}df$

(body)
$$\frac{\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \mathsf{e} \rangle \gg \longrightarrow_P \ll \chi', \gamma', \langle \rho', \lambda', \Theta, \mathsf{e}' \rangle \gg}{\ll \chi, \gamma, \langle \rho'', \lambda, \Theta, \mathbf{return}^\eta(\rho, \mathsf{e}) \rangle \gg \longrightarrow_P \ll \chi', \gamma', \langle \rho'', \lambda', \Theta, \mathbf{return}^\eta(\rho', \mathsf{e}') \rangle \gg}$$

Figure 3: Operational semantics of $\mathcal{F}ickle_{\mathtt{MT}}$: some $\longrightarrow_P$ reduction rules

- *(Extended) Expressions*, $\mathsf{e} \in \mathbf{E}$, defined by adding the clauses "$\mid\ \iota\ \mid\ \mathbf{return}^\eta(\rho, \mathsf{e})$" to the pseudo-grammar defining expressions (in Figure 1), where ι is an address, $\eta \in \{\mathsf{o}, \mathsf{i}\}$, and ρ is a frame (defined below).

- *(Extended) Values*, $\mathsf{v} \in \mathbf{Val} \subseteq \mathbf{E}$, defined by adding the clause "$\mid\ \iota$" to the pseudo-grammar defining values (in Figure 1), where ι is an address. So the set of *(extended) values* is $\{\mathsf{true}, \mathsf{false}, \mathtt{null}\} \cup \mathbf{I}$.

- *Objects*, $\mathbf{o} \in \mathbf{O} \triangleq (\mathbf{FN} \rightarrow_{\mathbf{fin}} \mathbf{Val}) \times \mathbf{CN}$, i.e., pairs of finite mappings from *field names*, in $\mathbf{FN}$, to values and *class names*, in $\mathbf{CN}$, denoted by $[[\mathsf{f}_1 : \mathsf{v}_1, \ldots, \mathsf{f}_\mathsf{r} : \mathsf{v}_\mathsf{r}]]^\mathsf{c}$. By $\mathbf{o}[\mathsf{f} : \mathsf{v}]$ we denote the object such that $\mathbf{o}[\mathsf{f} : \mathsf{v}](\mathsf{f}) = \mathsf{v}$ and $\mathbf{o}[\mathsf{f} : \mathsf{v}](\mathsf{f}') = \mathbf{o}(\mathsf{f}')$, for $\mathsf{f}' \neq \mathsf{f}$.

- *Heaps*, $\chi \in \mathbf{H} \triangleq \mathbf{I} \rightarrow_{\mathbf{fin}} \mathbf{O}$, i.e., finite mappings from addresses to objects, denoted by $[\iota_1 \mapsto \mathbf{o}_1, \ldots, \iota_\mathsf{n} \mapsto \mathbf{o}_\mathsf{n}]$. As for objects we use $\chi[\iota : \mathbf{o}]$ to denote the heap such that $\chi[\iota : \mathbf{o}](\iota) = \mathbf{o}$ and $\chi[\iota : \mathbf{o}](\iota') = \chi(\iota')$, for $\iota' \neq \iota$.

- *Frames*, $\rho \in \mathbf{F} \triangleq (\{\mathsf{x}\} \rightarrow \mathbf{Val}) \bigcup (\{\mathtt{this}\} \rightarrow \mathbf{I})$, which are mappings from the parameter x to values, and from the metavariable $\mathtt{this}$ to addresses. For denoting the update of ρ we use the same conventions as for heaps.

- *Global Objects State*, $\gamma \in \mathbf{GOS} \triangleq \mathbf{I} \rightarrow \{-1\} \cup \mathbb{N}$, i.e., mappings from addresses to integers greater than or equal to -1. If $\gamma(\iota) \geq 0$ then exactly $\gamma(\iota)$ threads could use the object at address ι as method receivers of methods which cannot re-classify the object at address ι. If $\gamma(\iota) = -1$ then exactly one thread executes one or more methods on the object at address ι: at least one of these methods could re-classify the object at address ι. For denoting the update of γ we use the same conventions as for heaps.

- *Local Objects State*, $\lambda \in \mathbf{LOS} \triangleq \mathbf{I} \rightarrow \{-1, 0, 1\}$, i.e., mappings from addresses to the set $\{-1, 0, 1\}$. If $\lambda(\iota) = 1$ then the current thread could use the object at address ι as method receivers of methods which cannot re-classify the object at address ι. If $\lambda(\iota) = 0$ then the current thread is executing no method on the object at address ι. If $\lambda(\iota) = -1$ then the current thread is executing one or more methods on the object at ad-

$$\frac{\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \mathsf{e} \rangle \gg \longrightarrow_P \ll \chi', \gamma', \langle \rho', \lambda', \Theta', \mathsf{e}' \rangle \gg}{\ll \chi, \gamma, \{..., \langle \rho, \Theta, \lambda, \mathsf{e} \rangle, ...\} \gg \longmapsto_P \ll \chi', \gamma', \{..., \langle \rho', \lambda', \Theta', \mathsf{e}' \rangle, ...\} \gg} \text{(step)}$$

$$\frac{}{\ll \chi, \gamma, \{..., \langle \rho, \lambda, \Theta, \mathcal{C}[\mathbf{spawn}(\mathsf{e})] \rangle, ...\} \gg \longmapsto_P \ll \chi, \gamma, \{..., \langle \rho, \lambda, \Theta, \mathcal{C}[\mathsf{true}] \rangle, \langle \rho, \lambda_0, \langle \{\,\}, \{\,\} \rangle, \mathsf{e} \rangle, ...\} \gg} \text{(spawn}^{\mathsf{o}}\text{)}$$

$$\frac{}{\ll \chi, \gamma, \{..., \langle \rho, \lambda, \Theta, \mathcal{E}[\mathbf{return}^{\eta}(\rho', \mathcal{C}[\mathbf{spawn}(\mathsf{e})])] \rangle, ...\} \gg \longmapsto_P \ll \chi, \gamma, \{..., \langle \rho, \lambda, \Theta, \mathcal{E}[\mathbf{return}^{\eta}(\rho', \mathcal{C}[\mathsf{true}])] \rangle, \langle \rho', \lambda_0, \langle \{\,\}, \{\,\} \rangle, \mathsf{e} \rangle, ...\} \gg} \text{(spawn}^{\mathsf{i}}\text{)}$$

$$\frac{}{\ll \chi, \gamma, \{..., \langle \rho, \lambda_0, \langle \{\,\}, \{\,\} \rangle, \mathsf{v} \rangle, ...\} \gg \longmapsto_P \ll \chi, \gamma, \{..., ...\} \gg} \text{(val)} \qquad \frac{}{\ll \chi, \gamma_0, \{\} \gg \longmapsto_P \texttt{end}} \text{(end)}$$

$$\frac{\ll \chi, \gamma, \langle \rho, \lambda, \Theta, \mathsf{e} \rangle \gg \longrightarrow_P \texttt{exc}}{\ll \chi, \gamma, \{..., \langle \rho, \lambda, \Theta, \mathcal{E}[\mathsf{e}] \rangle, ...\} \gg \longmapsto_P \texttt{exc}} \text{(exc)}$$

Figure 4: Operational semantics of $\mathcal{F}ickle_{\mathtt{MT}}$: $\longmapsto_P$ reduction rules

dress ι: at least one of these methods could re-classify the object at address ι. For denoting the update of λ we use the same conventions as for heaps.

- *Effects*, $\Theta = \langle \phi, \psi \rangle \in \mathbf{EF} \triangleq (\mathcal{P}_{fin}(\mathbf{CN}) \times \mathcal{P}_{fin}(\mathbf{CN}))$ are pairs of finite sets of class names. The first component of the pair, ϕ, is the *re-classification effect* and the second, ψ, is the *receive effect*. Inclusion and union of effects are defined component-wise:

$$\langle \phi, \psi \rangle \subseteq \langle \phi', \psi' \rangle \iff \phi \subseteq \phi' \text{ and } \psi \subseteq \psi'$$
$$\langle \phi, \psi \rangle \cup \langle \phi', \psi' \rangle = \langle \phi \cup \phi', \psi \cup \psi' \rangle.$$

- *Redexes*, $r \in \mathbf{R} \subseteq \mathbf{E}$ ::=

$$\begin{array}{l} \mathbf{if}\ \mathsf{v}\ \mathbf{then}\ \mathsf{e}\ \mathbf{else}\ \mathsf{e} \\ \mid \mathsf{v};\mathsf{e} \mid \mathbf{new}\ \mathsf{c} \mid \mathtt{this} \mid \mathsf{x}{:=}\mathsf{v} \mid \iota.\mathsf{f}{:=}\mathsf{v} \\ \mid \mathsf{x} \mid \iota.\mathsf{f} \mid \iota.\mathsf{m}(\mathsf{v}) \mid \mathbf{return}^{\eta}(\rho, \mathsf{e}) \mid \mathtt{this}\Downarrow\mathsf{c} \\ \mid \mathtt{null}.\mathsf{f}{:=}\mathsf{v} \mid \mathtt{null}.\mathsf{f} \mid \mathtt{null}.\mathsf{m}(\mathsf{v}) \end{array}$$

- *Evaluation Contexts*, $\mathcal{C} \in \mathbf{C}$ ::=

$$\begin{array}{l} [\,] \mid \mathbf{if}\ \mathcal{C}\ \mathbf{then}\ \mathsf{e}\ \mathbf{else}\ \mathsf{e} \mid \mathcal{C};\mathsf{e} \mid \mathsf{x}{:=}\mathcal{C} \\ \mid \mathcal{C}.\mathsf{f}{:=}\mathsf{e} \mid \iota.\mathsf{f}{:=}\mathcal{C} \mid \mathcal{C}.\mathsf{f} \mid \mathcal{C}.\mathsf{m}(\mathsf{e}) \mid \iota.\mathsf{m}(\mathcal{C}) \\ \mid \mathtt{null}.\mathsf{f}{:=}\mathcal{C} \mid \mathtt{null}.\mathsf{m}(\mathcal{C}) \end{array}$$

- *Extended Evaluation Contexts*, $\mathcal{E} \in \mathbf{EC}$ defined by adding the clause "$\mid \mathbf{return}^{\eta}(\rho, \mathcal{E})$", where $\eta \in \{\mathsf{o}, \mathsf{i}\}$, to the pseudo-grammar defining evaluation contexts.

A key observation is that we have an implicit stack of frames, since the $\mathbf{return}^{\eta}(\rho, \mathsf{e})$ expressions can be nested. An alternative formulation with an explicit stack is possible, but we claim that with our choice the rule (body) for evaluating method bodies is simpler. (See the explanation of the rule (body) below.)

The initial configuration for evaluating the expression e is

$$\ll \chi_0, \gamma_0, \{\langle \rho_0, \lambda_0, \langle \{\}, \{\} \rangle, \mathsf{e} \rangle\} \gg$$

where χ_0 is the empty heap, $\rho_0(\mathsf{x}) = \rho_0(\mathtt{this}) = \mathtt{null}$, $\gamma_0(\iota) = \lambda_0(\iota) = 0$ for all ι in the current heap,[3] and $\langle \{\}, \{\} \rangle$ is the empty effect.

[3]Clearly in the initial configuration γ_0 and λ_0 are the empty functions, but our definition allows to use them also in the creation of new threads and in final configurations.

For method calls we distinguish between method calls at the top level, rule (call$^{\mathsf{o}}$), and inner method calls, rule (call$^{\mathsf{i}}$). A method call is at the top level if and only if the effect in the current configuration is empty.

A *top level method call* on an object at the address ι can be executed only when:

1. the objects that could be re-classified by the execution of the method are not receivers of method calls in other threads, and

2. the object at the address ι, and the objects that the method execution could call methods on are not currently re-classified in other threads.

In the heap χ the set of the addresses of the objects which could be re-classified by a method execution is given by:

$$\phi \star_P \chi \;=\; \{\iota' \mid \chi(\iota') = [[...]]^{\mathsf{c}}\ \&\ \mathsf{c} \sqsubseteq_P \mathsf{c}' \text{ for some } \mathsf{c}' \in \phi\}$$

where ϕ is the re-classification effect of the method and $\mathsf{c} \sqsubseteq_P \mathsf{c}'$ means that c is a subclass of c'. Note that c is a subclass of a root class, and therefore it is either a root or a state class.

The set of the addresses of the objects which could be re-classifiable and receive methods during the execution of a method with receive effect ψ is given by:

$$\psi \star_P \chi \;=\; \{\iota' \mid \chi(\iota') = [[...]]^{\mathsf{c}}\ \&\ \mathsf{c} \text{ is a state or root class } \&\ \mathsf{c} \sqsubseteq_P \mathsf{c}' \text{ for some } \mathsf{c}' \in \psi\}$$

Therefore conditions 1. and 2. above can be formalized as:

$$\forall \iota' \in \Phi\ (\gamma(\iota') = 0) \qquad \text{and} \qquad \forall \iota' \in \Psi\ (\gamma(\iota') \geq 0)$$

where $\Phi = \phi \star_P \chi$ and $\Psi = \psi \star_P \chi \cup \{\iota\} - \Phi$.

At the beginning of the method call the global and the local objects states are updated by putting $\gamma(\iota') = \lambda(\iota') = -1$ for all the addresses $\iota' \in \Phi$ and $\gamma(\iota') = \gamma(\iota') + 1$, $\lambda(\iota') = 1$ for all the addresses $\iota' \in \Psi$.

The new effect is the effect of the method body with the addition of the receiver of the current call.

Instead, *inner method calls* do not change neither the effect nor γ, or λ: this is sound since the re-classification and receiver effects of inner calls are subsets of those of the corresponding top level calls.

For both top level and inner method calls the expression that is produced is a return expression, $\mathbf{return}^{\eta}(\rho, \mathsf{e})$, where η indicates the type of call (either top level or inner call), ρ is the evaluation frame for the body of the call, and e is the body of the method. The frame binds the formal parameter to the value of the actual parameter, v, and `this` to the receiver of the call, ι.

Symmetrically the return from a top level call, rule (ret$^{\text{o}}$), restores the initial values of γ, λ and Θ, while the return from inner calls, rule (ret$^{\text{i}}$), leaves them unchanged.

Rule (new) uses the effect in the current thread configuration to determine the values of γ and λ for the address ι of the newly created object: if the object could be re-classified then $\gamma(\iota) = \lambda(\iota) = -1$, if the object could receive a call without being re-classified then $\gamma(\iota) = \lambda(\iota) = 1$, otherwise $\gamma(\iota) = \lambda(\iota) = 0$.

For re-classification expressions, $\mathtt{this}\Downarrow\mathsf{c}$, 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 c. We preserve the fields belonging to the root superclass of c' and initialize the other fields of c according to their types. We use the term $\mathcal{R}(P, \mathsf{c})$ for denoting the least superclass of c which is not a state class: If c is a state class, then $\mathcal{R}(P, \mathsf{c})$ is its unique root superclass, otherwise $\mathcal{R}(P, \mathsf{c}) = \mathsf{c}$. Moreover, $\mathcal{F}s(P, \mathsf{c})$ denotes the set of fields defined in class c, $\chi(\iota)(\mathsf{f})$ the value of the field f in the object at address ι, and $\mathcal{F}(P, \mathsf{c}, \mathsf{f})$ the type of field f in class c.

The updating of frames and heaps is done respectively in rules ($:=_{\mathsf{v}}$) and ($:=_{\mathsf{f}}$).

The evaluation of method bodies is done by means of rule (body). The $\mathbf{return}^{\eta}(\rho, \mathsf{e})$ expression is an evaluation context, for the expression e, containing, in addition to the expression also the frame in which the expression has to be evaluated. The expression, e', and the frame ρ' resulting from the reduction of e in ρ are recorded in the resulting **return** expression, and the side effects of the evaluation of e on the heap and on the global and local object states are propagated in the returned configuration. Notice that only rules (call$^{\text{o}}$) and (ret$^{\text{o}}$) modify the effect, and these rules cannot be used for evaluating an expression inside a **return**. Therefore the application of rule (body) leaves the effect unchanged.

Rule (step) in Fig. 4 allows to use the sequential evaluation inside a multi-threaded configuration.

The new threads are generated by rules (spawn$^{\text{o}}$) and (spawn$^{\text{i}}$), outside and inside method calls respectively. Both rules start a new thread for the evaluation of the expression e, which initially has empty effects and initial local object state, while the expression $\mathbf{spawn}(\mathsf{e})$ is replaced with the constant `true`, so the evaluation of the expression containing the spawn expression can proceed in the old thread. The evaluation of the expression e is started in the current evaluation frame of the expression $\mathbf{spawn}(\mathsf{e})$. This allows the two threads to share objects in the heap. In particular, this is the case if the value bound to the variable x is an address. If x is bound to a boolean, its value can be considered an input value for the new thread. Moreover, if $\mathbf{spawn}(\mathsf{e})$ is well-typed e must be typed in an environment in which `this` is bound to Object, so no access to fields and methods of the current receiver can be done during the evaluation of e in the new thread (see rule (*spawn*) in Figure 5) . The normal termination is dealt with by rules (val) and (end), while rule (exc) propagates `exc`.

4. TYPING

In this section we illustrate the type and effect system of $\mathcal{F}ickle_{\mathtt{MT}}$. Besides the introduction of the receive effects, the main novelty with respect to [6] is that the typing rules allow re-classifications only inside compositions, in other words re-classification is a statement instead of an expression. We claim that this choice gives a cleaner use of re-classification making the programs easier to read. All the examples of $\mathcal{F}ickle$ programs in [6, 7, 5, 12, 1, 8, 4] satisfy this requirement.

We first present some interesting typing rules for expressions and then we discuss well-formed classes and programs.

The typing rules for expressions we will consider are given in Figure 5.

A first use of the information about the class of the receiver appears in rule (*seq*). The second expression, e', is typed in the updated environment $\Gamma[\mathtt{this}\mapsto\mathsf{c}]$ where c is the class of `this` after the evaluation of the first expression, e. So, the effect of the composition is the union of the effects of the components.

In rule (*cond*), the two branches may cause two different re-classifications for `this`, *i.e.* c_1 and c_2. So, after the evaluation we can only assert that `this` belongs to the least upper bound $\mathsf{c}_1 \sqcup_P \mathsf{c}_2$ of c_1 and c_2 with respect to the subclass hierarchy in the program P.

Consider rule (*meth*): the evaluation of the method's body could modify the class of `this` in the calling expression. This could happen if a superclass of the class of `this` in the calling expression is among the re-classification effects of the called method. (Existence of such a class implies uniqueness, since effects are sets of root classes.) For taking this into account, we define the application of re-classification effects to classes:

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

For method calls we lookup (using the function $\mathcal{M}(P, \mathsf{m}, \mathsf{c})$) the definition of method m in the class c of the receiver. Moreover we add the class c of the receiver to the receive effect.

The re-classification $\mathtt{this}\Downarrow\mathsf{c}$ is type correct if the environment Γ is well formed, i.e. it is of the shape $\Gamma = \{\mathsf{t}\ \mathsf{x}, \mathsf{c}\ \mathtt{this}\}$, where t is either **bool** or a non-state class and c is any class (this is expressed by the condition $P \vdash \Gamma\ \diamond$). Moreover c, the target of the re-classification, must be a root or state class (this is expressed by the condition $P \vdash \mathsf{c}\ \diamond_{rs}$), and c and the class of `this` before the mutation (the class $\Gamma(\mathtt{this})$) must be subclasses of the same root class. The re-classification effect is $\{\mathcal{R}(P, \mathsf{c})\}$.

In rule (*spawn*) we assume that `this` has type Object in the environment used to type the expression which will be evaluated in the new thread. The aim is to avoid that the new thread uses in a specific way the object referred by `this` in the main thread before the spawning.

We avoid re-classifications inside an expression by asking that its re-classification effect does not contain the least superclass of the class of `this` which is not a state class, i.e. by conditions like $\mathcal{R}(P, \Gamma(\mathtt{this})) \notin \phi$.

A program is well formed if the inheritance hierarchy is well-formed and all its classes are well-formed. 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

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

$$\frac{P,\mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{bool} \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi,\psi\rangle \qquad \mathcal{R}(P,\mathsf{\Gamma}(\mathtt{this})) \notin \phi \qquad P,\mathsf{\Gamma} \vdash \mathsf{e}_1 : \mathsf{t} \parallel \mathsf{c}_1 \parallel \Theta_1 \qquad P,\mathsf{\Gamma} \vdash \mathsf{e}_2 : \mathsf{t} \parallel \mathsf{c}_2 \parallel \Theta_2}{P,\mathsf{\Gamma} \vdash \mathbf{if}\ \mathsf{e}\ \ \mathbf{then}\ \ \mathsf{e}_1\ \ \mathbf{else}\ \mathsf{e}_2 : \mathsf{t} \parallel \mathsf{c}_1 \sqcup_P \mathsf{c}_2 \parallel \langle\phi,\psi\rangle \cup \Theta_1 \cup \Theta_2}\ (cond)$$

$$\frac{P,\mathsf{\Gamma} \vdash \mathsf{e}_0 : \mathsf{c} \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi_0,\psi_0\rangle \qquad P,\mathsf{\Gamma} \vdash \mathsf{e}_1 : \mathsf{t}_1 \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi_1,\psi_1\rangle \qquad \mathcal{R}(P,\mathsf{\Gamma}(\mathtt{this})) \notin \phi_0 \cup \phi_1 \qquad \mathcal{M}(P,\mathsf{c},\mathsf{m}) = \mathsf{t}\ \mathsf{m}(\mathsf{t}_1\ \mathsf{x})\ \langle\phi,\psi\rangle\ \{\ ...\ \}}{P,\mathsf{\Gamma} \vdash \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1) : \mathsf{t} \parallel \phi@_P\mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi,\psi\rangle \cup \langle\phi_0,\psi_0\rangle \cup \langle\phi_1,\psi_1\rangle \cup \langle\{\},\{\mathsf{c}\}\rangle}\ (meth)$$

$$\frac{P \vdash \mathsf{\Gamma}\ \diamond \qquad P \vdash \mathsf{c}\ \diamond_{rs} \qquad \mathcal{R}(P,\mathsf{c}) = \mathcal{R}(P,\mathsf{\Gamma}(\mathtt{this}))}{P,\mathsf{\Gamma} \vdash \mathtt{this}{\Downarrow}\mathsf{c} : \mathsf{c} \parallel \mathsf{c} \parallel \langle\{\,\mathcal{R}(\mathsf{P},\mathsf{c})\,\},\{\}\rangle}\ (recl)$$

$$\frac{P,\{\mathsf{t}_1\ \mathsf{x}, \mathsf{Object}\ \mathtt{this}\} \vdash \mathsf{e} : \mathsf{t} \parallel \mathsf{Object} \parallel \Theta}{P,\{\mathsf{t}_1\ \mathsf{x}, \mathsf{c}\ \mathtt{this}\} \vdash \mathbf{spawn}(\mathsf{e}) : \mathsf{bool} \parallel \mathsf{c} \parallel \langle\{\},\{\}\rangle}\ (spawn)$$

$$\frac{P,\mathsf{\Gamma} \vdash \mathsf{x} : \mathsf{t} \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\{\},\{\}\rangle \qquad P,\mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{t} \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi,\psi\rangle \qquad \mathcal{R}(P,\mathsf{\Gamma}(\mathtt{this})) \notin \phi}{P,\mathsf{\Gamma} \vdash \mathsf{x}{:=}\mathsf{e} : \mathsf{t} \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi,\psi\rangle}\ (a\text{-}var)$$

$$\frac{P,\mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{c} \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi,\psi\rangle \qquad P,\mathsf{\Gamma} \vdash \mathsf{e}' : \mathsf{t} \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi',\psi'\rangle \qquad \mathcal{R}(P,\mathsf{\Gamma}(\mathtt{this})) \notin \phi \cup \phi' \qquad \mathcal{F}(P,\mathsf{c},\mathsf{f}) = \mathsf{t}}{P,\mathsf{\Gamma} \vdash \mathsf{e}.\mathsf{f}{:=}\mathsf{e}' : \mathsf{t} \parallel \mathsf{\Gamma}(\mathtt{this}) \parallel \langle\phi,\psi\rangle \cup \langle\phi',\psi'\rangle}\ (a\text{-}field)$$

Figure 5: Some typing rules for expressions

of that of the overridden method. 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.

5. SOUNDNESS

In order to discuss soundness we need to consider agreements between programs, values, types, environments, heaps, frames and re-classification effects. We denote by $P,\mathsf{\Gamma} \vdash \chi,\rho\,\diamond$ the agreements between the environment $\mathsf{\Gamma}$, the heap χ and the frame ρ with respect to the program P. The notation $P,\phi \vdash \chi \lhd \chi'$ means that the classes of objects in the heap χ' can be obtained from those in the heap χ by applying the re-classification effect ϕ.

In our system, to state *subject reduction* we need typing rules also for the extended expressions, i.e. for ι, **return**(), and `exc`. The typing of addresses depends on the heap, and typing of return expressions depends on the typing of the frame inside the return expression.

We first generalize the notion of environment: An *extended environment* can contain assumption of the shape $\mathsf{c}\ \iota$ (to be read “the object at the address ι is of class c”).

Definition 1. *The* principal environment *relative to a program P, a heap χ and a frame ρ (notation $\Gamma_{(P,\chi,\rho)}$) is the extended environment defined by:*

$$\Gamma_{(P,\chi,\rho)}(\mathsf{x}) = \begin{cases} \mathbf{bool} & \textit{if } \rho(\mathsf{x}) \in \{\mathsf{true},\mathsf{false}\} \\ \mathcal{R}(P,\mathsf{c}) & \textit{if } \chi(\rho(\mathsf{x})) = [\![...]\!]^{\mathsf{c}} \end{cases}$$
$$\Gamma_{(P,\chi,\rho)}(\mathtt{this}) = \mathsf{c}\ \textit{if}\ \chi(\rho(\mathtt{this})) = [\![...]\!]^{\mathsf{c}}$$
$$\Gamma_{(P,\chi,\rho)}(\iota) = \mathsf{c}\ \textit{if}\ \chi(\iota) = [\![...]\!]^{\mathsf{c}}.$$

We have that $P,\Gamma_{(P,\chi,\rho)} \vdash \chi,\rho\,\diamond$. The key property of principal environments is:

Lemma 1. $P,\mathsf{\Gamma} \vdash \mathsf{e} : \mathsf{t} \parallel \mathsf{c} \parallel \Theta$ *and* $P,\mathsf{\Gamma} \vdash \chi,\rho\,\diamond$ *imply* $P,\Gamma_{(P,\chi,\rho)} \vdash \mathsf{e} : \mathsf{t} \parallel \mathsf{c} \parallel \Theta$.

The typing rules for extended expressions are given in Figure 6. The type of an address is the class of the object at that address. The expression `exc` has an arbitrary type: by $P \vdash \mathsf{c}\ \diamond$ we mean that the class c is well-formed in program P. The type of $\mathbf{return}^{\eta}(\rho',\mathsf{e})$ in the principal basis relative to ρ is the type of the expression e in the principal basis relative to ρ'. The final class of `this` is obtained by applying the re-classification effect of e to its initial class.

For stating subject reduction we have to consider separately the sequential reduction and the creation of a new thread.

Theorem 1. *[Subject Reduction for $\longrightarrow_P$] Let*

- $P,\Gamma_{(P,\chi,\rho)} \vdash \mathsf{e} : \mathsf{t} \parallel \mathsf{c} \parallel \langle\phi,\psi\rangle$, *and*
- $\ll \chi,\gamma,\langle\rho,\lambda,\Theta,\mathsf{e}\rangle \gg \longrightarrow_P \ll \chi',\gamma',\langle\rho',\lambda',\Theta',\mathsf{e}'\rangle \gg$

then

- $P,\Gamma_{(P,\chi',\rho')} \vdash \mathsf{e}' : \mathsf{t} \parallel \mathsf{c} \parallel \langle\phi,\psi\rangle$, *and*
- $P,\phi \vdash \chi \lhd \chi'$.

Theorem 2. *[Subject Reduction for* **spawn**()*] Let*

- $P,\mathsf{\Gamma} \vdash \mathcal{C}[\mathbf{spawn}(\mathsf{e})] : \mathsf{t} \parallel \mathsf{c} \parallel \Theta$,

then for some t',Θ'*:*

- $P,\mathsf{\Gamma} \vdash \mathcal{C}[\mathtt{true}] : \mathsf{t} \parallel \mathsf{c} \parallel \Theta$,
- $P,\mathsf{\Gamma}' \vdash \mathsf{e} : \mathsf{t}' \parallel \mathsf{Object} \parallel \Theta'$,

where $\mathsf{\Gamma}' = \{\mathsf{\Gamma}(\mathsf{x})\ \mathsf{x}, \mathsf{Object}\ \mathtt{this}\}$.

We end this section by stating that each multi-threaded configuration obtained by starting from a typable expression is deadlock free.

Definition 2. *A multi-threaded configuration Δ is* reachable for a well-formed program P, *notation $P \vdash \Delta\diamond$, if there is a typable expression e such that*

$$\ll \chi_0, \gamma_0, \{\langle\rho_0,\lambda_0,\langle\{\},\{\}\rangle,\mathsf{e}\rangle\} \gg \longmapsto_P^{*} \Delta.$$

Theorem 3. *[Progress] If $P \vdash \Delta\diamond$ and $\Delta \notin \{\mathtt{end},\mathtt{exc}\}$ then $\Delta \longmapsto_P \Delta'$ for some Δ'.*

The main property implying progress is that inner method calls have effects that are contained in the effects of the outer calls. This property can be formulated as follows:

$$\frac{\Gamma_{(P,\chi,\rho)}(\iota) = \mathsf{c}}{P, \Gamma_{(P,\chi,\rho)} \ \vdash \ \iota \ : \ \mathsf{c} \parallel \Gamma_{(P,\chi,\rho)}(\mathtt{this}) \parallel \langle\{\},\{\}\rangle} \ (addr) \qquad \frac{\mathsf{t{=}bool} \text{ or } (\mathsf{t{=}c} \text{ and } P \vdash \mathsf{c} \ \diamond)}{P, \Gamma_{(P,\chi,\rho)} \ \vdash \ \mathtt{exc} \ : \ \mathsf{t} \parallel \Gamma_{(P,\chi,\rho)}(\mathtt{this}) \parallel \langle\{\},\{\}\rangle} \ (exc)$$

$$\frac{P, \Gamma_{(P,\chi,\rho')} \ \vdash \ \mathsf{e} \ : \ \mathsf{t} \parallel \mathsf{c}' \parallel \langle\phi,\psi\rangle}{P, \Gamma_{(P,\chi,\rho)} \ \vdash \ \mathbf{return}^{\eta}(\rho', \mathsf{e}) \ : \ \mathsf{t} \parallel \phi @_P \Gamma_{(P,\chi,\rho)}(\mathtt{this}) \parallel \langle\phi,\psi\rangle} \ (ret)$$

Figure 6: Typing rules for extended expressions

LEMMA 2. *Let*

- $\mathcal{M}(P, \mathsf{c}, \mathsf{m}) = \mathsf{t} \ \mathsf{m}(\mathsf{t}_1 \ \mathsf{x}) \ \Theta \ \{ \ \mathsf{e} \ \}$,
- $\mathsf{e} = \mathcal{C}[\mathsf{e}_0.\mathsf{m}'(\mathsf{e}_1)]$,
- c_0 *the class of* e_0 *in the derivation of* P, t_1 x, c $\mathtt{this}$ $\vdash$ $\mathsf{e} : \mathsf{t} \parallel \mathsf{c}'' \parallel \Theta$,
- $\mathcal{M}(P, \mathsf{c}_0, \mathsf{m}') = \mathsf{t}' \ \mathsf{m}'(\mathsf{t}'_1 \ \mathsf{x}) \ \Theta' \ \{ \ \mathsf{e}' \ \}$,

then $\Theta' \cup \langle\{\},\{\mathsf{c}_0\}\rangle \subseteq \Theta$.

Remark 1. For the sequential reduction $\longrightarrow_P$ Theorem 3 implies the standard progress which says that a well-typed expression is either a value or it reduces. That is, if:

$$P, \Gamma_{(P,\chi,\rho)} \ \vdash \ \mathsf{e} \ : \ \mathsf{t} \parallel \mathsf{c} \parallel \Theta$$

$$\text{and } P \vdash \ll \chi, \gamma, \{\langle\rho, \lambda, \Theta, \mathsf{e}\rangle\} \gg \diamond$$

where e is not a value, then

$$\ll \chi, \gamma, \langle\rho, \lambda, \Theta, \mathsf{e}\rangle \gg \longrightarrow_P \ll \chi', \gamma', \langle\rho', \lambda', \Theta', \mathsf{e}'\rangle \gg$$

for some χ', γ', ρ', λ', Θ', e'.

6. CONCLUSION

The present paper shows how re-classification can fit in a multi-threaded language. This is obtained by combining a static type and effect system with a synchronization mechanism based on effect information. We conjecture that similar combinations could be useful for preventing deadlocks in presence of (suitably restricted) constructs for standard synchronization.

It is cumbersome for programmers to annotate each method with a receive effect that has to estimate all the classes of objects that may be the receivers of method calls during the execution of the body of the method. We can improve this in several ways.

- By allowing *state classes to override only methods defined in root or state classes.* In this way receive effects could contain only root and state classes. This would significantly reduce the size of receive effects and, moreover, would allow method that do not deal with re-classification to have empty receive effect (such methods already have empty re-classification effect).

- By introducing *a keyword to specify that a class cannot be extended with a root class.* Again this would reduce the size of the receive effects, because such a class, and its subclasses, do not need to be mentioned in the receive effects of a method. Indeed, variables of such classes would never be bound to re-classifiable objects.

To simplify the presentation we have defined global and local states for all addresses, but clearly they are used only when the objects belong to state or root classes. This observation can be used to implement efficiently our synchronization mechanism.

7. REFERENCES

[1] C. Anderson. Implementing Fickle, Imperial College, final year thesis, June 2001.

[2] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structual operational semantics of multi-threaded java. In J. Alves-Foss, editor, *Formal Syntax and Semantics Of Java*, volume 1523 of *LNCS*, pages 157–200. Springer, 1999.

[3] C. Chambers. Predicate Classes. In O. Nierstrasz, editor, *ECOOP'93*, volume 707 of *LNCS*, pages 268–296. Springer, 1993.

[4] F. Damiani, S. Drossopoulou, and P. Giannini. Refined effects for unanticipated object re-classification: Fickle3 (extended abstract). In *ICTCS'03*, LNCS 2841, pages 97–10. Springer, 2003.

[5] S. Drossopoulou. Three Case Studies in $\mathcal{F}ickle_{\mathrm{II}}$. Technical report, Imperial College, 2002.

[6] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic Object Re-classification. In J. L. Knudsen, editor, *ECOOP'01*, volume 2072 of *LNCS*, pages 130–149. Springer, 2001.

[7] 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.

[8] D. Fidgett. Extending $\mathcal{F}ickle_{\mathrm{II}}$, Imperial College, final year thesis, June 2002.

[9] G. Ghelli and D. Palmerini. Foundations of Extended Objets with Roles (*extended abstract*). In *FOOL6*, 1999.

[10] C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 290:741–778, 2003.

[11] M. Serrano. Wide Classes. In R. Guerraoui, editor, *ECOOP'99*, volume 1628 of *LNCS*, pages 391–415. Springer, 1999.

[12] A. Shuttlewood. Implementing $\mathcal{F}ickle_{\mathrm{II}}$ on the JVM, Imperial College, final year thesis, June 2002.

[13] A. Tailvasaari. Object Oriented Programming with Modes. *Journal of Object Oriented Programming*, 6(3):27–32, 1993.

[14] K. Wight and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.