JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 11
Special issue: OOPS track at SAC 2004, Nicosia/Cyprus

On Re-classification and Multi-threading

Ferruccio Damiani, Dip. di Informatica, Univ. di Torino, Italy
Mariangiola Dezani-Ciancaglini, Dip. di Informatica, Univ. di Torino, Italy
Paola Giannini, Dip. di Informatica, Univ. del Piemonte Orientale, Italy

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 $\mathcal{F}ickle_{\mathsf{MT}}$. 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 access to non-existing members of objects. We show that in the execution of a well-typed expression such delays do not produce deadlock. Lastly we discuss a translation from $\mathcal{F}ickle_{\mathsf{MT}}$ into Java, showing how the operational semantics can be implemented with the standard Java multi-threading constructs.

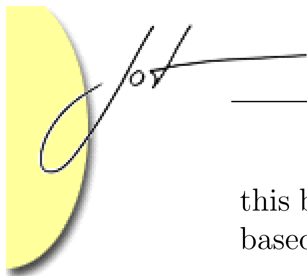
1 INTRODUCTION

Re-classifiable objects support the modification of object's behavior by changing class membership at runtime, see e.g. [6, 15, 13, 17, 10, 11, 8]. The language $\mathcal{F}ickle$ [10, 11, 8] 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 $\mathcal{F}ickle_{\mathsf{MT}}$.

In $\mathcal{F}ickle_{\mathsf{MT}}$ 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 $\mathcal{F}ickle$ [10, 11, 8] 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

Cite this article as follows: Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, Paola Giannini: "On Re-classification and Multi-threading", in *Journal of Object Technology*, vol. 3, no. 11, December 2004, Special issue: OOPS track at SAC 2004, Nicosia/Cyprus, pp. 5–30, http://www.jot.fm/issues/issue_2004_12/article1



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 re-classifiable objects that may receive messages. The operational semantics uses the 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.* [5, 14]), instead, consider a semantics closer to the implementation.

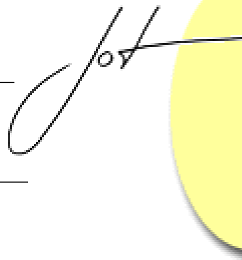
To investigate the practicality of the ideas around $\mathcal{F}ickle_{\mathsf{MT}}$ we provide a translation of $\mathcal{F}ickle_{\mathsf{MT}}$ into Java [4]. The translation extends a previous translation of $\mathcal{F}ickle$ into Java, see [1], and its improved version [2]. To implement the delays needed in method calls we use the standard synchronization constructs of Java. The translation preserves the behavior of expressions.

This paper is organized as follows: Section 2 provides a brief overview of $\mathcal{F}ickle_{\mathsf{MT}}$. Section 3 presents the operational semantics and Section 4 introduces the type and effect system. In Section 5 a translation from $\mathcal{F}ickle_{\mathsf{MT}}$ into Java is defined. The first four sections of the present paper are a revised and improved version of [7]: in particular the present operational semantics and effect system are simpler than those in [7].

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

$\mathcal{F}ickle_{\mathsf{MT}}$ 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 $\mathcal{F}ickle_{\mathsf{MT}}$ 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.



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

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 $\mathcal{F}ickle_{\mathsf{MT}}$

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

The expression $\mathsf{this}{\Downarrow}\mathsf{c}$ 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).[1]

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 defined in state classes are called *state methods*. Calls to state methods are traced since, due to the usual scoping rules, these methods are the only methods that can access fields defined in state classes.

Methods have only one parameter and are declared by

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

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

[1] In $\mathcal{F}ickle_{\mathsf{MT}}$ (following $\mathcal{F}ickle$ [10, 11, 8]), state classes are alternative rather than cumulative, as for example roles in [13]. We do not see major technical problems in considering a different design choice that, in the spirit of roles, requires re-classification $\mathsf{this}{\Downarrow}\mathsf{c}$ to keep the value of the fields from the previous time (if any) the re-classified object was in the state class c. The investigation of this issue is outside of the scope of the present paper.

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 root classes which conservatively estimates the set of classes whose objects could receive calls of state methods, but cannot be re-classified during the evaluation of e (this implies $\phi \cap \psi = \emptyset$).

The receive effect ψ, used only by the synchronization mechanism, plays a crucial role in guaranteeing safety in presence of multi-threading. For a program to be well-formed, we require that, if $\langle\phi, \psi\rangle$ is the effect of a state method defined in a state class, which is a subclass of the root class c, then $\mathsf{c} \in \phi \cup \psi$. This ensures that, during the execution of a method m, if an object of a state class is the receiver of a state method, then its root super-class occurs in the effect of the method m. In this way we can assure that an object is never re-classified by one thread when another thread uses it as receiver of a state method. The above condition is not enforced in [10, 11, 8], where receive effects are not considered. Note that the effects of state methods are always non-empty, while the effects of non-state methods can be either empty or non-empty. In particular, non-state methods that do not re-classify objects and do not call state methods can have empty effect.

The expression **spawn**(e) causes the execution of the expression e in a new thread. Since the spawned expression e is no more inside the original method body, we require that this does not occur in e.

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 $\{\mathsf{t}\ \mathsf{x},\ \mathsf{c}\ \mathsf{this}\}$; 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

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

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

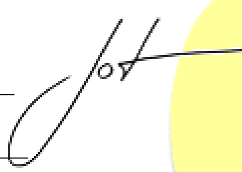
An Example

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

In the typing judgements of $\mathcal{F}ickle_{\mathsf{MT}}$, as in [10], the class c and the *re-classification effect* ϕ are used to track how the receivers of methods can change class. For example, if $\Gamma(\mathsf{this})=\mathsf{Frog}$ and $\Gamma'(\mathsf{this})=\mathsf{Prince}$ we have (see the typing rules in Figure 5):

$$P_{\mathsf{pl}}, \Gamma \vdash \mathsf{this.pouch}{:=}\ \mathsf{null} : \mathsf{Vocal} \parallel \mathsf{Frog} \parallel \langle\{\},\{\}\rangle$$
$$P_{\mathsf{pl}}, \Gamma \vdash \mathsf{this}{\Downarrow}\mathsf{Prince} : \mathsf{Prince} \parallel \mathsf{Prince} \parallel \langle\{\mathsf{Player}\},\{\}\rangle$$
$$P_{\mathsf{pl}}, \Gamma' \vdash \mathsf{this.sword}{:=}\ \mathbf{new}\ \mathsf{Weapon} : \mathsf{Weapon} \parallel \mathsf{Prince} \parallel \langle\{\},\{\}\rangle$$

[2] This example is a multi-threaded variant of an example proposed in [11].



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

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

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

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

state class Prince extends Player{
   Weapon sword;
   bool wake( )⟨{ }, { Player } ⟩{this.sword.swing( ); this.brave}
   Weapon kissed( )⟨{ }, { Player } ⟩{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 body of 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 field not found error. This is realized taking into account that, for Γ(this)=Game and Γ(x)=Player, the two method calls have typing

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

and typing

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

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( ).

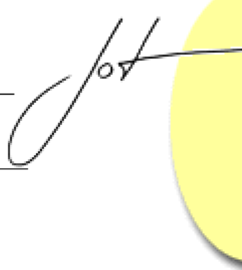
To avoid these problems, we require that *each object that is re-classified by one thread is not concurrently re-classified or used as receiver of state methods.* This requirement is enforced by introducing a mapping, γ, from root class names to integers. The mapping γ, called "re-classification counter", identifies the objects that may be re-classified and the objects that may receive messages in the active methods. Before executing a method call the rules of the operational semantics check (using the information given by the re-classification counter γ and by the method effect) a sufficient condition for the above requirement and delay the execution of the method call until it is 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 access to non-existing members of objects.

3 OPERATIONAL SEMANTICS

The semantics of $\mathcal{F}ickle_{\mathsf{MT}}$ is a small step operational semantics, presented in the style advocated in [18]. To model multi-threaded computations we consider *multi-threaded configurations* composed by exactly one heap, one "re-classification counter" (which is shared by all the threads) and one set of tuples of *single-threaded configurations* ⟨stack, effect, expression⟩ (one tuple for each thread). The "re-classification counter" (whose structure is detailed at page 12) and the "effects" of all threads keep the information about "which threads are active on which re-classifiable objects". Notice that, according to $\mathcal{F}ickle_{\mathsf{MT}}$ syntax, a thread may access fields defined in a state class only if it is executing a state method call.

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



$$(\text{call}^{\mathbf{s}}) \quad \ll \chi, \gamma, \langle \zeta, \Theta, \iota.\mathsf{m}(\mathsf{v}) \rangle \gg \longrightarrow_P \ll \chi, \gamma, \langle \zeta \cdot [\mathsf{x} \mapsto \mathsf{v}, \mathsf{this} \mapsto \iota], \Theta, \mathbf{return^s}(\mathsf{e}) \rangle \gg$$
$$\text{if} \quad \chi(\iota) = [\![...]\!]^{\mathsf{c}},\ \mathcal{M}(P, \mathsf{c}, \mathsf{m}) = \mathsf{t\ m(\ t_1\ x)}\ \langle \phi, \psi \rangle\ \{\ \mathsf{e}\ \}, \text{ and}$$
$$\text{either } |\zeta| \geq 2 \text{ or } \langle \phi, \psi \rangle = \langle \{\}, \{\} \rangle$$

$$(\text{call}^{\mathbf{n}}) \quad \ll \chi, \gamma, \langle \zeta, \Theta, \iota.\mathsf{m}(\mathsf{v}) \rangle \gg \longrightarrow_P \ll \chi, \gamma', \langle \zeta \cdot [\mathsf{x} \mapsto \mathsf{v}, \mathsf{this} \mapsto \iota], \langle \phi, \psi \rangle, \mathbf{return^n}(\mathsf{e}) \rangle \gg$$
$$\text{if} \quad \chi(\iota) = [\![...]\!]^{\mathsf{c}},\ \mathcal{M}(P, \mathsf{c}, \mathsf{m}) = \mathsf{t\ m(\ t_1\ x)}\ \langle \phi, \psi \rangle\ \{\ \mathsf{e}\ \},$$
$$|\zeta| = 1,\ \langle \phi, \psi \rangle \neq \langle \{\}, \{\} \rangle,$$
$$\forall \mathsf{c}' \in \phi(\gamma(\mathsf{c}') = 0) \text{ and } \forall \mathsf{c}' \in \psi(\gamma(\mathsf{c}') \geq 0)$$
$$\text{where} \quad \gamma' = \gamma[\mathsf{c}' : -1 \mid \mathsf{c}' \in \phi][\mathsf{c}' : \gamma(\mathsf{c}') + 1 \mid \mathsf{c}' \in \psi]$$

$$(\text{ret}^{\mathbf{s}}) \quad \ll \chi, \gamma, \langle \zeta \cdot \rho, \Theta, \mathbf{return^s}(\mathsf{v}) \rangle \gg \longrightarrow_P \ll \chi, \gamma, \langle \zeta, \Theta, \mathsf{v} \rangle \gg$$

$$(\text{ret}^{\mathbf{n}}) \quad \ll \chi, \gamma, \langle \zeta \cdot \rho, \langle \phi, \psi \rangle, \mathbf{return^n}(\mathsf{v}) \rangle \gg \longrightarrow_P \ll \chi, \gamma', \langle \zeta, \langle \{\}, \{\} \rangle, \mathsf{v} \rangle \gg$$
$$\text{where} \quad \gamma' = \gamma[\mathsf{c} : 0 \mid \mathsf{c} \in \phi][\mathsf{c} : \gamma(\mathsf{c}) - 1 \mid \mathsf{c} \in \psi]$$

$$(\text{new}) \quad \ll \chi, \gamma, \langle \zeta, \Theta, \mathbf{new}\ \mathsf{c} \rangle \gg \longrightarrow_P \ll \chi', \gamma, \langle \zeta, \Theta, \iota \rangle \gg$$
$$\text{if} \quad \chi(\iota) \text{ is undefined}$$
$$\text{where} \quad \mathcal{F}s(P, \mathsf{c}) = \{\mathsf{f}_1, ..., \mathsf{f}_\mathsf{r}\},\ \ \forall \mathsf{l} \in 1, ..., \mathsf{r} : \mathsf{v_l} \text{ initial for } \mathcal{F}(P, \mathsf{c}, \mathsf{f_l}), \text{ and}$$
$$\chi' = \chi[\iota \mapsto [\![\mathsf{f}_1 : \mathsf{v}_1, ..., \mathsf{f_r} : \mathsf{v_r}]\!]^{\mathsf{c}}]$$

$$(\text{rec}) \quad \ll \chi, \gamma, \langle \sigma \cdot \rho, \Theta, \mathsf{this} \Downarrow \mathsf{c} \rangle \gg \longrightarrow_P \ll \chi[\iota \mapsto [\![\mathsf{f}_1 : \mathsf{v}_1, ..., \mathsf{f_{r+q}} : \mathsf{v_{r+q}}]\!]^{\mathsf{c}}], \gamma, \langle \sigma \cdot \rho, \Theta, \iota \rangle \gg$$
$$\text{where} \quad \iota = \rho(\mathsf{this}), \chi(\iota) = [\![...]\!]^{\mathsf{c}'},\ \mathcal{F}s(P, \mathcal{R}(\mathsf{P}, \mathsf{c})) = \{\mathsf{f}_1, ..., \mathsf{f_r}\},$$
$$\forall \mathsf{l} \in 1, ..., \mathsf{r} : \mathsf{v_l} = \chi(\iota)(\mathsf{f_l}),\ \mathcal{F}s(P, \mathsf{c}) \setminus \{\mathsf{f}_1, ..., \mathsf{f_r}\} = \{\mathsf{f_{r+1}}, ..., \mathsf{f_{r+q}}\}, \text{ and}$$
$$\forall \mathsf{l} \in \mathsf{r} + 1, ..., \mathsf{r} + \mathsf{q} : \mathsf{v_l} \text{ initial for } \mathcal{F}(P, \mathsf{c}, \mathsf{f_l})$$

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

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

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

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

$$\longmapsto_P \ \subseteq \ \Big(\mathbf{H} \times \mathbf{Cnt} \times \mathcal{P}_{fin}(\mathbf{S} \times \mathbf{Eff} \times \mathbf{E})\Big) \times \Big((\mathbf{H} \times \mathbf{Cnt} \times \mathcal{P}_{fin}(\mathbf{S} \times \mathbf{Eff} \times \mathbf{E})) \cup \{\mathsf{exc}, \mathsf{end}\}\Big)$$

which is defined in terms of another reduction relation specifying how a sequential reduction step rewrites the heap, the re-classification counter, and exactly one single-threaded configuration:

$$\longrightarrow_P \ \subseteq \ \Big(\mathbf{H} \times \mathbf{Cnt} \times (\mathbf{S} \times \mathbf{Eff} \times \mathbf{R})\Big) \times \Big((\mathbf{H} \times \mathbf{Cnt} \times (\mathbf{S} \times \mathbf{Eff} \times \mathbf{E})) \cup \{\mathsf{exc}\}\Big)$$

A well-typed program terminates either normally with the special term end or with the special term exc modeling 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:

$$\frac{\ll \chi, \gamma, \langle \zeta, \Theta, r \rangle \gg \longrightarrow_P \ll \chi', \gamma', \langle \zeta', \Theta', \mathsf{e}' \rangle \gg}{\ll \chi, \gamma, \{..., \langle \zeta, \Theta, \mathcal{C}[r] \rangle, ...\} \gg \longmapsto_P \ll \chi', \gamma', \{..., \langle \zeta', \Theta', \mathcal{C}[\mathsf{e}'] \rangle, ...\} \gg} \text{(step)}$$

$$\frac{}{\ll \chi, \gamma, \{..., \langle \sigma \cdot \rho, \Theta, \mathcal{C}[\mathbf{spawn}(\mathsf{e})] \rangle, ...\} \gg \longmapsto_P \ll \chi, \gamma, \{..., \langle \sigma \cdot \rho, \Theta, \mathcal{C}[\mathsf{true}] \rangle, \langle \rho', \Theta_0, \mathsf{e} \rangle, ...\} \gg} \text{(spawn)}$$

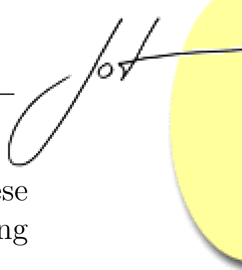
where $\rho' = \rho[\mathsf{this} : \mathsf{null}]$ and $\Theta_0 = \langle \{\}, \{\} \rangle$

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

$$\frac{}{\ll \chi, \gamma_0, \{\} \gg \longmapsto_P \mathsf{end}} \text{(end)} \qquad \frac{\ll \chi, \gamma, \langle \zeta, \Theta, r \rangle \gg \longrightarrow_P \mathsf{exc}}{\ll \chi, \gamma, \{..., \langle \zeta, \Theta, \mathcal{C}[r] \rangle, ...\} \gg \longmapsto_P \mathsf{exc}} \text{(exc)}$$

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

- *Addresses*, $\iota \in \mathbf{I}$ (we assume a denumerable set of addresses).
- *(Extended) Expressions*, $\mathsf{e} \in \mathbf{E}$, defined by adding the clauses "$\mid \ \iota \ \mid \ \mathbf{return}^{\kappa}(\mathsf{e})$" to the pseudo-grammar defining expressions (in Figure 1), where ι is an address and $\kappa \in \{\mathbf{s}, \mathbf{n}\}$.
- *(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 $\{\mathsf{true}, \mathsf{false}, \mathsf{null}\} \cup \mathbf{I}$.
- *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 $[\![\mathsf{f}_1 : \mathsf{v}_1, ..., \mathsf{f}_r : \mathsf{v}_r]\!]^{\mathsf{c}}$. 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. 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*, $\rho \in \mathbf{F} \triangleq \{\mathsf{x}, \mathsf{this}\} \rightarrow \mathbf{Val}$, which are mappings from the parameter x to values, and from the metavariable this to addresses or null. For denoting the update of ρ we use the same conventions as for heaps.
- *Stacks*, $\sigma \in \mathbf{S} \triangleq \bigcup_{n \in \mathbf{N}} \mathbf{F}^n$, which are finite sequences of frames $\rho_1 \cdot \cdots \cdot \rho_n$ $(n \geq 0)$, where ρ_1 is the bottom of the stack and ρ_n is the top of the stack. Non-empty stacks are ranged over by ζ.
- *Re-classification counters*, $\gamma \in \mathbf{Cnt} \triangleq \mathbf{RCN} \rightarrow \{-1\} \cup \mathbf{N}$, i.e., mappings from *root class names*, in $\mathbf{RCN}$, to integers greater than or equal to -1. If $\gamma(\mathsf{c}) \geq 0$ then exactly $\gamma(\mathsf{c})$ threads could use objects belonging to (a subclass of) c as receivers of state methods which cannot re-classify object belonging to (a subclass of) c. If $\gamma(\mathsf{c}) = -1$ then exactly one thread executes one or



more methods on objects belonging to (a subclass of) c: at least one of these methods could re-classify objects belonging to (a subclass of) c. For denoting the update of γ we use the same conventions as for heaps.

- *Effects*, $\Theta = \langle\phi,\psi\rangle \in \mathbf{Eff} \triangleq \{\langle\phi',\psi'\rangle \in (\mathcal{P}_{fin}(\mathbf{RCN}) \times \mathcal{P}_{fin}(\mathbf{RCN})) \mid \phi' \cap \psi' = \{\,\}\}$ are pairs of disjoint finite sets of root class names. The first component of the pair, ϕ, is the *re-classification effect* and the second, ψ, is the *receive effect. Union of effects* is defined component-wise preserving the condition that the intersection of the re-classification component and of the receive component of the effect must be empty:

$$\langle\phi,\psi\rangle \cup \langle\phi',\psi'\rangle \;=\; \langle\phi\cup\phi', (\psi\cup\psi') - (\phi\cup\phi')\rangle.$$

 Since $\langle\phi,\psi\rangle$ is a sub-effect of $\langle\phi',\psi'\rangle$ iff there is $\langle\phi'',\psi''\rangle$ such that

$$\langle\phi,\psi\rangle \cup \langle\phi'',\psi''\rangle \;=\; \langle\phi',\psi'\rangle,$$

 we get the following definition of *inclusion between effects*:

$$\langle\phi,\psi\rangle \subseteq \langle\phi',\psi'\rangle \quad \text{if} \quad \phi\subseteq\phi' \text{ and } \psi\cup\phi\subseteq\psi'\cup\phi'.$$

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

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

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

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

The *initial configuration* for evaluating the expression e in a program P is

$$\ll \chi_0, \gamma_0, \{\langle\rho_0, \Theta_0, \mathsf{e}\rangle\} \gg$$

where χ_0 is the empty heap, $\gamma_0(\mathsf{c}) = 0$ for all root classes c which occur in P, $\rho_0(\mathsf{x}) = \rho_0(\mathsf{this}) = \mathsf{null}$, and $\Theta_0 = \langle\{\},\{\}\rangle$.

A typical configuration of one thread is:

$$\langle\rho_1 \cdot \cdots \cdot \rho_n, \Theta, \mathcal{C}[\textbf{return}^{\kappa_1}(...(\textbf{return}^{\kappa_n}(\mathsf{e}))...)]\rangle$$

where $\mathcal{C}$ and e do not contain occurrences of **return**, the outermost **return** corresponds to the bottom frame ρ_1, and the innermost **return** corresponds to the top frame ρ_n.

Before discussing the reduction rules we introduce some notation.

The function $\mathcal{M}(P,\mathsf{m},\mathsf{c})$ gives the definition of method m in the class c.

The term $\mathcal{R}(P, \mathsf{c})$ denotes 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 $\mathcal{R}(P, \mathsf{c}) = \mathsf{c}$.

For method calls we distinguish between *standard and non-standard method calls*, rules ($\text{call}^{\mathbf{s}}$) and ($\text{call}^{\mathbf{n}}$). A standard method call, rule ($\text{call}^{\mathbf{s}}$), can always be executed and changes neither the statically declared effect Θ nor the re-classification counter γ. Instead, a non-standard method call, rule ($\text{call}^{\mathbf{n}}$), could be executed only when:

1. the objects that will be re-classified by the execution of the method are not receivers of state method calls in other threads, and
2. the objects that the method execution will call state methods on and, when the method is state, the receiver object are not currently re-classified in other threads.

The objects that will be re-classified by the execution of the method call must belong to (a subclass of) a class in the re-classification effect ϕ of the method. The objects that will receive state methods during the execution of a method call (including the receiver of the method call itself when the method is a state method) and will not be re-classified during the execution of the method call must belong to (a subclass of) the receiver effect ψ of the method. Therefore conditions 1. and 2. above are implied by the following conditions:

$$\forall \mathsf{c}' \in \phi \; (\gamma(\mathsf{c}') = 0) \qquad \text{and} \qquad \forall \mathsf{c}' \in \psi \; (\gamma(\mathsf{c}') \geq 0).$$

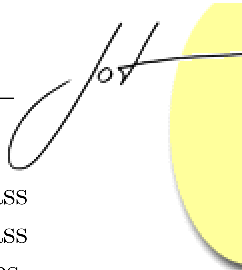
At the beginning of the method call the effect is set to $\langle \phi, \psi \rangle$ and the re-classification counter is updated by putting $\gamma(\mathsf{c}') = -1$ for all the root classes $\mathsf{c}' \in \phi$ and $\gamma(\mathsf{c}') = \gamma(\mathsf{c}') + 1$ for all the root classes $\mathsf{c}' \in \psi$.

In well-formed programs (see page 17), the effects of inner method calls (i.e., the method calls done inside another a method call) are sub-effects of those of the corresponding top level call. Therefore, *non-standard method calls are exactly top-level calls of methods such that the set* $\phi \cup \psi$ *is not empty.* The current stack is never empty and it contains exactly one frame in all and only the top level evaluations. This justifies the conditions $|\zeta| \geq 2$ and $|\zeta| = 1$ (where $|\zeta|$ denotes the number of frames in the stack ζ) in rules ($\text{call}^{\mathbf{s}}$) and ($\text{call}^{\mathbf{n}}$).

Both standard and non-standard method calls push the evaluation frame for the body of the call on the stack. The frame binds the formal parameter to the value of the actual parameter, v, and this to the receiver of the call, ι. The method call is rewritten into a return expression, $\mathbf{return}^{\kappa}(\mathsf{e})$, where κ indicates the kind of the method call ($\mathbf{s}$ for standard and $\mathbf{n}$ for non-standard) and e is the body of the method.

The return from a standard method call, rule ($\text{ret}^{\mathbf{s}}$), leaves the effect and the re-classification counter unchanged. Instead, the return from a non-standard method call, rule ($\text{ret}^{\mathbf{n}}$), restores the value $\langle \{\}, \{\} \rangle$ for the effect and updates the value of the re-classification counter.

Rule (new) creates a new object of class c at an address ι. The term $\mathcal{F}s(P, \mathsf{c})$ denotes the set of fields defined in class c and $\mathcal{F}(P, \mathsf{c}, \mathsf{f})$ the type of field f in class c.



For re-classification expressions, $\mathsf{this}\Downarrow\mathsf{c}$, the object bound to this, which is of class c', is replaced by a new object of class c. The fields belonging to the root superclass of c' are preserved and the other fields of c are initialized according to their types. The term $\chi(\iota)(\mathsf{f})$ denotes the value of the field f in the object at address ι.

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

Rule (step) in Figure 4 allows to perform a sequential reduction step inside (one of the single-threaded configurations composing) the multi-threaded configuration.

Rule (spawn) generates a new thread for the evaluation of the expression e, which initially has empty effect, while the expression **spawn**(e) is replaced with the constant true, so the evaluation of the expression containing the spawn expression can proceed in the old thread. The stack of the new thread generated for the evaluation of the expression e contains only one frame in which x is bound to the current value in the old thread and this is bound to null. The two threads can share objects in the heap 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. The value of this is bound to null since the spawned expression does not contain occurrences of this (see page 8).

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 $\mathcal{F}ickle_{\mathsf{MT}}$. Besides the introduction of the receive effects, the main novelty with respect to [10] is that the typing rules allow the object bound to this to be re-classified only by expression occurring in "statement position" (i.e., not by: the test of a conditional, the receiver and the parameter of a method call, the left-hand side and the right-hand side of an assignment,...). 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 [10, 11, 9, 16, 3, 12, 8] 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 $\Gamma[\mathsf{this}{\mapsto}\mathsf{c}]$ 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. The condition $\mathcal{R}(P, \Gamma(\mathsf{this})) \not\in \phi$ (the re-classification effect ϕ

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

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

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

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

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

Figure 5: Some typing rules for expressions

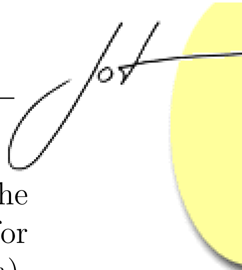
does not contain the least root superclass of the class of this) ensures that the object bound to this is not re-classified by the test part of the conditional.

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:

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

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, if the class c of the receiver is a state or a root class such that $\mathcal{R}(P, \mathsf{c})$ does not occur in the re-classification effect, we add the class $\mathcal{R}(P, \mathsf{c})$ to the receive effect. To this aim we define the term $\mathcal{E}(P, \mathsf{c})$ as the set $\{\mathcal{R}(P, \mathsf{c})\}$ if c is a state or a root class, and as the empty set otherwise.

The re-classification $\mathsf{this} \Downarrow \mathsf{c}$ is type correct if the environment Γ is well-formed, i.e. it is of the shape $\Gamma = \{\mathsf{t\ x}, \mathsf{c\ this}\}$, 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 re-classification (the class Γ(this)) must be subclasses of the same root class. The re-classification effect is $\{\mathcal{R}(P, \mathsf{c})\}$.



In a well-formed program (see below) this cannot occur in an expression of the shape **spawn**(e). In spite of this, we need a type for this in the environment Γ for typing e inside **spawn**, since Γ(this) is used for example in rules (*cond*) and (*meth*). So in rule (*spawn*) we assume that this has type Object in the environment used to type the expression e. Recall that e will be evaluated in a new thread and the value of this in this new thread will be null.

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 *a method may redefine a superclass method only if* it has the same name, argument, and result type, and *its effect is a sub-effect of that of the overridden method*. Recall that $\langle\phi,\psi\rangle$ is a *sub-effect* of $\langle\phi',\psi'\rangle$ if $\phi\subseteq\phi'$ and $\psi\cup\phi\subseteq\psi'\cup\phi'$. For instance, $\langle\{\},\ \{\mathsf{Player}\}\ \rangle$ is a sub-effect of $\langle\ \{\mathsf{Player}\}\ ,\{\}\rangle$, so in the Example of Figure 2, the definitions of method cursed in class Frog, and of method kissed in class Prince are correct. Method bodies must be well-formed, must return a value appropriate for the method signature, and their effect must be a sub-effect of that in the signature. *Effects of state methods must contain the root superclass of the state class defining the state method.* Moreover, *the arguments of spawn expressions do not contain occurrences of* this.

We end this section by stating the soundness property of our operational semantics and type system: we can assure that starting from a well-typed expression we always obtain well-typed expressions and we never reach a deadlock. We need some definitions.

Definition 1 *1. An environment* Γ agrees with a program P, a heap χ and a frame ρ *if:*

$$\Gamma(\mathsf{x}) = \begin{cases} \mathbf{bool} & \textit{if } \rho(\mathsf{x})\in\{\mathsf{true},\mathsf{false}\} \\ \mathcal{R}(P,\mathsf{c_x}) & \textit{if } \chi(\rho(\mathsf{x})) = [\![\ldots]\!]^{\mathsf{c_x}} \end{cases}$$
$$\Gamma(\mathsf{this}) = \mathsf{c_{this}} \quad \textit{if } \chi(\rho(\mathsf{this})) = [\![\ldots]\!]^{\mathsf{c_{this}}}$$

2. *The expression* e *is* typable w.r.t. a program P, a heap χ and a frame ρ *if*

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

for some type t*, class* c*, effect* Θ *and environment* Γ *which agrees with* P*,* χ *and* ρ*.*

3. *A multi-threaded configuration* Δ *is* reachable for a well-formed program P*, notation* $P\vdash\Delta\diamond$*, if there is an expression* e *typable w.r.t.* P*,* χ_0 *and* ρ_0 *such that*

$$\ll\chi_0,\gamma_0,\{\langle\rho_0,\Theta_0,\mathsf{e}\rangle\}\gg\longmapsto_P^*\Delta.$$

where $\ll\chi_0,\gamma_0,\{\langle\rho_0,\Theta_0,\mathsf{e}\rangle\}\gg$ *is an initial configuration for* e *(defined at page 13).*

Note that Definition 1.2 requires the run-time value of the method argument and the receiver to have exactly the type specified in the type environment. This condition

can be relaxed by asking only that the method argument and/or the receiver has a subtype of the type specified in the environment. We choose the current definition to avoid introducing the subtype relation, since the resulting notions of typability coincide.

Theorem 1 (Type Preservation) *If* $P \vdash \ll \chi, \gamma, \{..., \langle \sigma \cdot \rho, \Theta, \mathsf{e} \rangle, ...\} \gg \diamond$ *then* e *is typable w.r.t.* P*,* χ*, and* ρ*.*[3]

As observed by a referee, the above formulation of type preservation is quite weak. A stronger formulation (not reported here, since it requires the introduction of further definitions) is given in [7].

Theorem 2 (Progress) *If* $P \vdash \Delta\diamond$ *and* $\Delta \notin \{\mathsf{end}, \mathsf{exc}\}$ *then* $\Delta \longmapsto_P \Delta'$ *for some* Δ'*.*

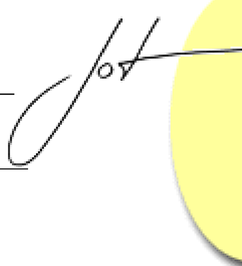
5 A TRANSLATION OF $\mathcal{F}ickle_{\mathsf{MT}}$ INTO JAVA

The translation we define is based on the translation of $\mathcal{F}ickle$ into Java described in [2] (revised and improved version of [1]). Following [2], in order to simplify the presentation, we consider as target language an idealized Java (in which, as in Java, declarations may have initializations and declarations and statements may be interleaved in a block), extended with *block-expressions*, which are blocks containing declarations and statements as regular blocks with a final expression. A block-expression is evaluated like a standard Java block-statement (that is from left to right) and, when it occurs in expression position, yields the value of the last expression. For example, see the translation of re-classification in Figure 11.

The translation mapping is defined by cases on the various syntactic constructs: programs, classes, field declarations, method declarations, and expressions. The main difference with respect to the translation given in [2] is the presence of the new translation mapping $[\![\cdot]\!]_{eTop}$, from $\mathcal{F}ickle_{\mathsf{MT}}$ expressions to Java, which has been introduced in order to correctly translate the expressions occurring as arguments of spawn expressions.

In order to be self-contained we describe the translation in full. To state a formal result of correctness of the present translation we would need to give an operational semantics of the subset of Java which is the target of the translation. Then, we should prove that any computation of an expression can be simulated by a computation of the translated expression and vice versa. This is outside the scope of the present paper.

[3] In general e may be an extended expression, so we need typing rules for extended expressions, as done in [7].



$$
\begin{aligned}
[\![P]\!] \triangleq\ & \textbf{class}\ \ \mathsf{Identity}\ \textbf{extends}\ \mathsf{Object}\ \{\ \mathsf{FickleObject\ imp;}\ \}\\
& \textbf{class}\ \mathsf{FickleObject}\ \textbf{extends}\ \mathsf{Object}\ \{\ \mathsf{Identity\ id;}\ \}\\
& \textbf{class}\ \ \mathsf{Gamma}\ \{\ \cdots\ \}\\
& \textbf{class}\ \ \mathsf{SpawnL}_1\ \{\ \cdots\ \}\\
& \cdots\cdots\\
& \textbf{class}\ \ \mathsf{SpawnL}_l\ \{\ \cdots\ \}\\
& [\![\mathit{class}_1]\!]_{\mathit{class}}(P)\\
& \cdots\cdots\\
& [\![\mathit{class}_n]\!]_{\mathit{class}}(P)
\end{aligned}
$$

where $P^{\mathrm{L}} = \mathit{class}_1 \cdots \mathit{class}_n$

Figure 6: Translation of programs

The classes FickleObject and Identity

Like the translation in [2], the translation from $\mathcal{F}ickle_{\mathsf{MT}}$ into Java is based on the idea that each object **o** is encoded by a pair $<$**id**, **imp**$>$ of objects; we call **id** the *identity object of* **imp** and **imp** the *implementor object of* **id**. Roughly speaking, **id** provides the identity of **o**, and **imp** the behavior of **o**. A re-classification of **o** changes **imp** but not **id**: method invocations are resolved by **imp**, whereas objects are accessed through **id**. Hence, two implementors paired with the same identity represent the same object at different execution stages.

Implementor objects belong to the class FickleObject which contains the field id which is of type Identity. Identity objects belong to the class Identity which contains the field imp which is of type FickleObject (see Figure 6 for the definition of the two classes). Variables that in the original program were of type class will be translated in variables of type Identity. All classes declared in a program are subclasses of FickleObject.

Translation of programs

A program, P is a sequence of class declarations: $\mathit{class}_1 \cdots \mathit{class}_n$. The translation mapping for programs, $[\![\cdot]\!]$, is defined on *labeled programs*. A labeled program P is a program where every occurrence of a **spawn** expression, **spawn**(e), has been marked with a distinguished label L (we consider a denumerable set of labels ranged over by L_1,L_2,...), producing a labeled **spawn** expression, $\textbf{spawn}^{\mathsf{L}}$(e). Thus, before applying the translation mapping $[\![\cdot]\!]$ on a program P, we have first to transform P into a labeled program, P^{L}, by labeling with fresh labels all the occurrences of **spawn** expressions in P.

The translation of a (labeled) program P contains the translation of the classes defined in the program plus a number of extra classes:

```
class SpawnL extends Thread {
   private theType(t) fieldX ;

   public SpawnL (theType(t) x) {
      fieldX=x ;
   }

   public void run ( ) {
      ([[e]]_eTop(P, {t x, Object this}))[fieldX/x];
   }
}
where:
t is the type of the parameter x in the method that contains spawn^L(e)
```

Figure 7: Classes SpawnL

- the classes FickleObject and Identity,
- classes SpawnL, one for each label L occurring in P^{L}, and
- the class Gamma, needed for bookkeeping.

We assume that there are no name clashes between these extra classes and the classes declared in P^{L}.

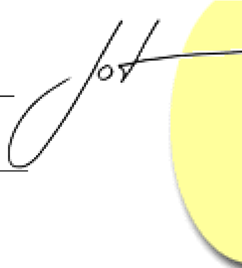
The translation of (labeled) programs is defined in Figure 6. The classes Identity and FickleObject have been discussed above, the classes SpawnL are defined in Figure 7, and Gamma is defined in Figures 8 and 9.

The classes SpawnL extend the Java library class Thread. Each class SpawnL has a field for storing a copy of the parameter of the method containing the spawn expression labeled L (field fieldX). The constructor initializes this field (see the translation of **spawn**$^{\mathsf{L}}$(e) in Figure 11). The correct type for the field fieldX is obtained applying the following operator (taken from [2]):

$$\mathit{theType}(\mathsf{t}) = \begin{cases} \mathsf{Identity} & \text{if } \mathsf{t} \text{ is a class} \\ \mathsf{t} & \text{otherwise} \end{cases} \tag{1}$$

to the current type t of the parameter.

The body of the method run is the translation through $[\![\cdot]\!]_{eTop}$ (see page 23) of the expression e which is the argument of **spawn**. As remarked in [2], since the translation of expressions depends on their types, both the program and the environment must be passed as argument to the corresponding translation functions. The environment used for the translation maps x to the type of the parameter of the method containing **spawn**$^{\mathsf{L}}$(e) and this to Object, in accord with the typing rule (*spawn*) of Figure 5. Recall that e does not contain occurrences of this. After the



```
class Gamma extends Object {
  private static Map gamma= new HashMap( );

  public static syncronized void preCall(Set phi, Set psi) {
    while(!mayCall(phi, psi)) {
         wait ();
    }
    addToGamma(phi, psi);
  }

  public static syncronized void postCall(Set phi, Set psi) {
    remFromGamma(phi, psi);
    notifyAll ();
  }

  private static bool mayCall(Set phi, Set psi) { ··· }

  private static void addToGamma(Set phi, Set psi) { ··· }

  private static void remFromGamma(Set phi, Set psi) { ··· }
}
```

Figure 8: Class Gamma

application of the translation mapping $[\![\cdot]\!]_{eTop}$ to e, the occurrences of x in $[\![\mathsf{e}]\!]_{eTop}$ are replaced with fieldX.

Notice that, at run-time, in a program generated by the present translation there are two kinds of objects, the objects extending FickleObject generated by the translation removing re-classification, and the SpawnL objects that contain the expressions inside the **spawn**: our threads. Objects of classes SpawnL are not receivers of any method call except for the start method of the Java library class Thread that executes the method run.

The class Gamma implements the re-classification counter γ (represented by the private static field gamma) and the computations needed before and after non-standard method calls. Since we want at most one thread at a time that modifies γ, the public methods of Gamma are synchronized. More precisely, the field gamma contains the mapping γ (defined at page 12) from root class names, represented as strings, to integers. The mapping γ is checked and updated in rules (call$^{\mathbf{n}}$) and (ret$^{\mathbf{n}}$) of Figure 3. The methods preCall and postCall of Figure 8 implement respectively what is done in these rules: they are synchronized and this assures that only one thread can read and modify the field gamma.

Rule (call$^{\mathbf{n}}$) allows the execution of the method call only if the condition

$$\forall \mathsf{c}' \in \phi(\gamma(\mathsf{c}') = 0) \text{ and } \forall \mathsf{c}' \in \psi(\gamma(\mathsf{c}') \geq 0) \tag{2}$$

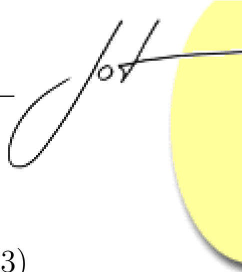
```
private static bool mayCall(Set phi, Set psi) {
   Iterator itphi= phi.iterator( );
   while (itphi.hasNext( )){
         String nextClass= (String)itphi.next( );
         if ((gamma.containsKey(nextClass)) && (gamma.get(nextClass)! = 0))
               return false ;
   }
   Iterator itpsi= psi.iterator( );
   while (itphi.hasNext( )){
         String nextClass= (String)itpsi.next( );
         if ((gamma.containsKey(nextClass)) && (gamma.get(nextClass) < 0))
               return false ;
   }
   return true ;
}

private static void addToGamma(Set phi, Set psi) {
   Iterator itphi= phi.iterator( );
   while (itphi.hasNext( )){
         gamma.put(itphi.next( ), −1);
   }
   Iterator itpsi= psi.iterator( );
   while (itphi.hasNext( )){
         String nextClass= (String)itpsi.next( );
         if (gamma.containsKey(nextClass))
               gamma.put(nextClass, gamma.get(nextClass) + 1);
         else gamma.put(nextClass, 1);
   }
}

private static void remFromGamma(Set phi, Set psi) {
   Iterator itphi= phi.iterator( );
   while (itphi.hasNext( )){
         gamma.put(itphi.next( ), 0);
   }
   Iterator itpsi= psi.iterator( );
   while (itphi.hasNext( )){
         String nextClass= (String)itpsi.next( );
         gamma.put(nextClass, gamma.get(nextClass) − 1);
   }
}
```

Figure 9: The private methods of class Gamma

is satisfied. The method mayCall of Figure 9 evaluates this condition. If condition



(2) is satisfied rule (call$^{\mathbf{n}}$) updates γ as follows:

$$\gamma[\mathsf{c}' : -1 \mid \mathsf{c}' \in \phi][\mathsf{c}' : \gamma(\mathsf{c}') + 1 \mid \mathsf{c}' \in \psi] \tag{3}$$

The method addToGamma of Figure 9 modifies gamma according to (3).

Rule (ret$^{\mathbf{n}}$) updates γ as follows:

$$\gamma[\mathsf{c} : 0 \mid \mathsf{c} \in \phi][\mathsf{c} : \gamma(\mathsf{c}) - 1 \mid \mathsf{c} \in \psi]$$

and method remFromGamma of Figure 9 implements this updating. The method postCall after calling remFromGamma notifies the change of gamma to all threads: this change may allow some threads to execute method calls.

For simplicity in the methods of Figure 9 we omit the wrapping of **int** into Integer and vice versa. Such wrappings are needed since the Java library class HashMap maps Object to Object.

Translation of classes, fields, and method declarations

Figure 10 gives the translation of classes, fields, and method declarations.

Each translated class extends class FickleObject and the translation preserves the inheritance hierarchy. Therefore it is useful to introduce the operator:

$$\mathit{theName}(\mathsf{c}) = \begin{cases} \mathsf{FickleObject} & \text{if } \mathsf{c} = \mathsf{Object} \\ \mathsf{c} & \text{otherwise} \end{cases}$$

Each $\mathcal{F}ickle_{\text{MT}}$ class c is translated into a single Java class containing the translation of the field and method declarations of c. The translation of these declarations is the same for any kind of class. For fields we just translate the type via the function $\mathit{theType}(\mathsf{t})$ defined in (1).

The program P, and the class c are passed as parameter to the translation function for methods, since the translation of the expression which is the method body depends on its typing judgement, and the typing judgement in turn depends on the program P, and on the current class, which is the type of this.

Translating a method consists of removing the effect annotation and translating the return type, the parameter type, and the body. To the translation function for the method body we pass the program P, the typing environment in which the current class is bound to this, and the type of the parameter to x.

Translation of expressions

For expressions we have two translation functions: $[\![\cdot]\!]_{expr}$ and $[\![\cdot]\!]_{eTop}$.

Starting from the translation $[\![\cdot]\!]_{expr}$ defined in [2] we:

$$\llbracket [\,\textbf{root} \mid \textbf{state}\,]\textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}'\ \{\ \mathsf{t}_1\ \mathsf{f}_1\ \cdots\ \mathsf{t}_n\ \mathsf{f}_n\ \mathit{meth}_1\ \cdots\ \mathit{meth}_m\ \}\rrbracket_{\mathit{class}}(P) \triangleq$$
$$\textbf{class}\ \mathsf{c}\ \ \textbf{extends}\ \ \mathit{theName}(\mathsf{c}')\ \{\ \mathit{theType}(\mathsf{t}_1)\ \mathsf{f}_1\ \cdots\ \mathit{theType}(\mathsf{t}_n)\ \mathsf{f}_n$$
$$\llbracket \mathit{meth}_1\rrbracket_{\mathit{meth}}(P,\mathsf{c})\ \cdots\ \llbracket \mathit{meth}_m\rrbracket_{\mathit{meth}}(P,\mathsf{c})$$
$$\}$$

$$\llbracket \mathsf{t}\ \mathsf{m}\ (\ \mathsf{t}'\ \mathsf{x})\ \Theta\ \ \{\,\mathsf{e}\}\rrbracket_{\mathit{meth}}(P,\mathsf{c}) \triangleq$$
$$\mathit{theType}(\mathsf{t})\ \mathsf{m}\ (\ \ \mathit{theType}(\mathsf{t}')\ \mathsf{x})\ \{\ \}\ \ \{\ \llbracket \mathsf{e}\rrbracket_{\mathit{expr}}(P,\{\mathsf{t}'\ \mathsf{x},\mathsf{c}\ \mathsf{this}\})\ \}$$

Figure 10: Translation of classes, fields, and method declarations

- extend $\llbracket\cdot\rrbracket_{expr}$ by adding the clause for **spawn** expressions (which was not in the language of [2]), and
- define $\llbracket\cdot\rrbracket_{eTop}$ as the "extended" $\llbracket\cdot\rrbracket_{expr}$ (i.e. as $\llbracket\cdot\rrbracket_{expr}$ in [2] plus the clause for **spawn** expressions) by modifying the translation of method calls to insert the checks and delays needed for non-standard calls.

The translation $\llbracket\cdot\rrbracket_{expr}$ is used for expressions in contexts in which we know that the method calls occurring in the expression will always be standard. It is applied to the bodies of the methods in the classes of the program, since the calls in them are always inner calls, and therefore are standard, see Figure 10. The translation $\llbracket\cdot\rrbracket_{eTop}$ is intended for expressions that could contain non-standard method calls. We use it for all the expressions in which there could be top level method calls. Such expressions are: the expressions inside **spawn**, which, according to our operational semantics, are evaluated at the top level, and the initial expression that we evaluate. The expressions inside **spawn** are the bodies of the run methods of SpawnL classes, see Figure 7.

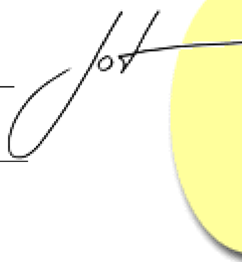
The translation $\llbracket\cdot\rrbracket_{expr}$

In Figure 11 we present the clauses of the translation $\llbracket\cdot\rrbracket_{expr}$. Most of the clauses are as in [2]. Recall that each object is encoded by a pair of objects <**id**, **imp**>, respectively with the fields id and imp pointing to each other and that the translation of an object is the corresponding **id** object. This means that in order to access the current implementation of an object we have to select the field imp in the object translation.

In field access, field update and method call, the cast of the field imp to *theName*(c) is needed since field imp has type FickleObject.

The clauses for field update and method call are simplified w.r.t. the corresponding clauses in [2]. The simplification comes from the fact that this can be re-classified only by expressions in "statement position".

Consider the clause for method call (which is similar to field update). First we evaluate the receiver and assign this value to the variable y. Recall that according



$[\![\mathsf{v}]\!]_{expr}(P,\Gamma) \triangleq \mathsf{v}$ $\qquad$ $[\![\mathsf{x}]\!]_{expr}(P,\Gamma) \triangleq \mathsf{x}$ $\qquad$ $[\![\mathsf{this}]\!]_{expr}(P,\Gamma) \triangleq \mathsf{this.id}$

$[\![\mathsf{e.f}]\!]_{expr}(P,\Gamma) \triangleq ((\mathit{theName}(\mathsf{c}))\ ([\![\mathsf{e}]\!]_{expr}(P,\Gamma).\mathsf{imp})).\mathsf{f}$ $\quad$ where $P,\Gamma \vdash \mathsf{e} : \mathsf{c} \parallel \mathsf{c}' \parallel \Theta$

$[\![\textbf{if } \mathsf{e} \textbf{ then } \mathsf{e}_1 \textbf{ else } \mathsf{e}_2]\!]_{expr}(P,\Gamma) \triangleq$
$\quad \textbf{if } [\![\mathsf{e}]\!]_{expr}(P,\Gamma) \textbf{ then } [\![\mathsf{e}_1]\!]_{expr}(P,\Gamma) \textbf{ else } [\![\mathsf{e}_2]\!]_{expr}(P,\Gamma)$

$[\![\mathsf{e}_1;\mathsf{e}_2]\!]_{expr}(P,\Gamma) \triangleq [\![\mathsf{e}_1]\!]_{expr}(P,\Gamma);[\![\mathsf{e}_2]\!]_{expr}(P,\Gamma[\mathsf{this}\mapsto\mathsf{c}])$ $\quad$ where $P,\Gamma \vdash \mathsf{e}_1 : \mathsf{t} \parallel \mathsf{c} \parallel \Theta$

$[\![\mathsf{x}{:=}\mathsf{e}]\!]_{expr}(P,\Gamma) \triangleq \mathsf{x}{:=}[\![\mathsf{e}]\!]_{expr}(P,\Gamma)$

$[\![\mathsf{e}_1.\mathsf{f}{:=}\mathsf{e}_2]\!]_{expr}(P,\Gamma) \triangleq \{$ $\mathsf{Identity\ y}= [\![\mathsf{e}_1]\!]_{expr}(P,\Gamma);$
$\quad \textbf{if } (\mathsf{y} == \mathsf{null})\ \textbf{throw new } \mathsf{nullPointerException}();$
$\quad (\mathit{theName}(\mathsf{c}))(\mathsf{y.imp}).\mathsf{f} = [\![\mathsf{e}_2]\!]_{expr}(P,\Gamma);$
$\}$ $\quad$ where $P,\Gamma \vdash \mathsf{e}_1 : \mathsf{c} \parallel \mathsf{c}' \parallel \Theta$

$[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_{expr}(P,\Gamma) \triangleq \{$ $\mathsf{Identity\ y}= [\![\mathsf{e}_1]\!]_{expr}(P,\Gamma);$
$\quad \textbf{if } (\mathsf{y} == \mathsf{null})\ \textbf{throw new } \mathsf{nullPointerException}();$
$\quad (\mathit{theName}(\mathsf{c}))(\mathsf{y.imp}).\mathsf{m}([\![\mathsf{e}_2]\!]_{expr});$
$\}$ $\quad$ where $P,\Gamma \vdash \mathsf{e}_1 : \mathsf{c} \parallel \mathsf{c}' \parallel \Theta$

$[\![\ \textbf{new } \mathsf{c}]\!]_{expr}(P,\Gamma) \triangleq \{$ $\mathit{theName}(\mathsf{c})\ \mathsf{theImp}= \textbf{new } \mathit{theName}(\mathsf{c});$
$\quad \mathsf{Identity\ theId}= \textbf{new } \mathsf{Identity};$
$\quad \mathsf{theImp.id}=\mathsf{theId};$
$\quad \mathsf{theId.imp}=\mathsf{theImp};$
$\quad \mathsf{theId};$
$\}$

$[\![\mathsf{this}\Downarrow\mathsf{c}]\!]_{expr}(P,\Gamma) \triangleq \{$ $\mathit{theName}(\mathsf{c})\ \mathsf{theImp}= \textbf{new } \mathit{theName}(\mathsf{c});$
$\quad \mathsf{Identity\ theId}=\mathsf{this.id};$
$\quad \mathcal{R}(P,\mathsf{c})\ \mathsf{theLastThis}= (\mathcal{R}(P,\mathsf{c}))\ (\mathsf{theId.imp});$
$\quad \mathsf{theImp.id}=\mathsf{theId};$
$\quad \mathsf{theId.imp}=\mathsf{theImp};$
$\quad \mathsf{theImp.f}_1=\mathsf{theLastThis.f}_1;$
$\quad \ldots$
$\quad \mathsf{theImp.f}_r=\mathsf{theLastThis.f}_r;$
$\quad \mathsf{theId};$
$\}$
where $\{\mathsf{f}_1,...,\mathsf{f}_r\}$ are the fields of the class $\mathcal{R}(P,\mathsf{c})$

$[\![\textbf{spawn}^{\mathsf{L}}(\mathsf{e})]\!]_{expr}(P,\Gamma) \triangleq \{\textbf{new } \mathsf{SpawnL(x).start(\,);\ true\ ;}\}$

Figure 11: The translation of expressions $[\![]\!]_{expr}$

Let c be such that $P, \Gamma \vdash e_1 : c \parallel c' \parallel \Theta'$ for some c' and Θ', and t, $\langle\{c_1, ..., c_m\}, \{d_1, ..., d_n\}\rangle$ be the return type and the effect of method m in class c.

$$
[\![e_1.m(e_2)]\!]_{eTop}(P,\Gamma) \triangleq
\begin{cases}
\{\ \mathsf{Identity}\ \ y{=}[\![e_1]\!]_{eTop}(P,\Gamma); \\
\quad \textbf{if}\ (y == \mathsf{null}) \\
\qquad \textbf{throw}\ \ \textbf{new}\ \ \mathsf{nullPointerException}(); \\
\quad (\mathit{theName}(c))(y.\mathsf{imp}).m([\![e_2]\!]_{eTop}(P,\Gamma)); \\
\}, & \text{if } m = n = 0 \\
\\
\{\ \mathit{theType}(t)\ \ \mathsf{result}; \\
\quad \mathsf{Identity}\ \ y{=}[\![e_1]\!]_{eTop}(P,\Gamma); \\
\quad \textbf{if}\ (y == \mathsf{null}) \\
\qquad \textbf{throw}\ \ \textbf{new}\ \ \mathsf{nullPointerException}(); \\
\quad \mathsf{Set}\ \ \mathsf{phi}{=}\,\textbf{new}\ \mathsf{HashSet.add}(\text{``}c_1\text{''})....\mathsf{add}(\text{``}c_m\text{''}); \\
\quad \mathsf{Set}\ \ \mathsf{psi}{=}\,\textbf{new}\ \mathsf{HashSet.add}(\text{``}d_1\text{''})....\mathsf{add}(\text{``}d_n\text{''}); \\
\quad \mathsf{Gamma.preCall}(\mathsf{phi},\mathsf{psi}); \\
\quad \mathsf{result}{=}\,(\mathit{theName}(c))(y.\mathsf{imp}).m([\![e_2]\!]_{eTop}(P,\Gamma)); \\
\quad \mathsf{Gamma.postCall}(\mathsf{phi},\mathsf{psi}); \\
\quad \mathsf{result}; \\
\}, & \text{otherwise}
\end{cases}
$$

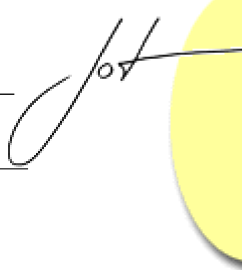
Figure 12: Translation of method calls for $[\![\cdot]\!]_{eTop}$

to the translation of [2] the values of expressions are either booleans or objects of class Identity. For this reason we declare y as an object of class Identity. If y is the null pointer we raise an exception. Otherwise, we call the method m of the object refereed to by $(\mathit{theName}(c))(y.imp)$ on the translation of the actual parameter e_2.

The translation of object creation creates an identity object and an implementor object and binds them to each other. The identity object is then returned.

For re-classification a new implementor object is created. All the fields of the root superclass of the previous implementor are copied into it, and this new implementor substitutes the old one.

We translate an occurrence of an expression $\mathbf{spawn}^{L}(e)$ by creating a new object of the class SpawnL. The argument of this constructor is the parameter x. The argument is assigned to the field fieldX (see the definition of class SpawnL in Figure 7). To start the thread we send the method start to the new object of class SpawnL and we return the value true in agreement with rule (spawn) of Figure 4. The method start is the method (of the Java library class Thread) that starts the execution of the method run.



```
class SpawnA extends Spawn {
  private Identity fieldX ;

  public SpawnA (Identity x) {
    fieldX=x ;
  }

  public void run ( ) { Identity result;
                        Identity y=fieldX;
                        if (y == null)  throw new nullPointerException();
                        Set phi=new Set( ).add("Player");
                        Set psi=new Set( );
                        Gamma.preCall(phi, psi);
                        result= (Player)(y.imp).kissed( );
                        Gamma.postCall(phi, psi);
                        result;
                      }

}
```

Figure 13: Class SpawnA

The translation $[\![\cdot]\!]_{eTop}$

All the clauses of the mapping $[\![\cdot]\!]_{eTop}$ except the one for method call are as the corresponding ones of $[\![\cdot]\!]_{expr}$. The clause for method call is given in Figure 12. This translation needs to take into account both call and return rules of Figure 3. We look at the class c of e_1 in the context of program P and environment Γ. Let $\langle\{\mathsf{c}_1, ..., \mathsf{c}_m\}, \{\mathsf{d}_1, ..., \mathsf{d}_n\}\rangle$ be the effect of method m in class c: if $m = n = 0$ then the translation is the same as in the mapping $[\![\cdot]\!]_{expr}$, since (call$^{\mathbf{s}}$) and (ret$^{\mathbf{s}}$) apply. Otherwise we have to use rules (call$^{\mathbf{n}}$) and (ret$^{\mathbf{n}}$). We declare the local variable result of type *theType*(t), where t is the return type of method m in class c, and we build two sets of strings (phi and psi) containing respectively the root class names (represented as strings) $\{\mathsf{c}_1, ..., \mathsf{c}_m\}$ and $\{\mathsf{d}_1, ..., \mathsf{d}_n\}$. As we already explained the method call Gamma.preCall() checks gamma and possibly changes it. Then, we execute the method call and when it ends we restore the initial conditions for gamma as described in rule (ret$^{\mathbf{n}}$).

Example

We sketch the translation of:

$$[\![\mathbf{spawn}^{\mathsf{A}}(\mathsf{x.kissed(\,)})]\!]_{eTop}(P_{\mathsf{pl}}, \Gamma)$$

where A is the label of the **spawn** expression, $\Gamma(\mathsf{this}) = \mathsf{Game}$, $\Gamma(\mathsf{x}) = \mathsf{Player}$ and program P_{pl} is given in Figure 2. The translation of this expression is:

```
{   new SpawnA(x).start( );
    true ;
}
```

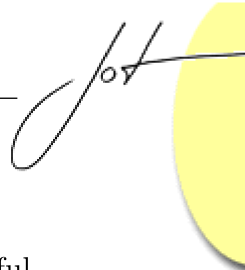
and the declaration of class SpawnA is given in Figure 13.

6 CONCLUSION

We have presented $\mathcal{F}ickle_{\mathsf{MT}}$, an extension of a multi-threaded class based language in which objects may change class at run-time. A combination of a static type and effect system with a mechanism delaying the execution of method calls allows re-classification to be safely used in presence of multi-threading. Moreover, the delays introduced do not cause deadlock. We have also presented a translation of $\mathcal{F}ickle_{\mathsf{MT}}$ into Java. The translation shows how to realize the non-standard method call/return rules using standard multi-threading constructs such as synchronized methods of Java.

The aim of this paper was to analyze the interaction between multi-threading and re-classification, so we considered a minimal extension of $\mathcal{F}ickle$ including a primitive **spawn**(e) starting the expression e in a new thread. We plan to apply our technique to the standard multi-threading present in Java, that is the definition of subclasses of Thread that are explicitly created and started. Moreover, we are working on one side at a proof of soundness for the current language that extends the sketch given in [7], and on the other at a proof of soundness of the translation given in this paper. For the latter we need to formalize the significant subset of Java used, which includes static members of classes, threads and synchronized methods.

As noticed by a referee, the present approach might unnecessarily limit multi-threading in some situations. For instance, if a thread is executing a method that will (or may) at some point re-classify *a given set of objects* of a (subclass of a) root class is r, then such a thread would block any other thread that attempts to execute a method that will (or may) invoke methods on *a disjoint set of objects* of a (subclass of) r. A possibility for overcoming this problem could be to use object-level locking (instead of class-level locking). However, we think that a better program design would be declaring different classes for the two sets of disjoint objects. We believe that, in order to exploit the potentiality of object-level locking, it is necessary to rely on dynamic information (instead of on a statically known approximation), and that guaranteeing absence of deadlocks (as Theorem 2 does for class based locking) leads to enforce limitations on multi-threading similar to the ones of this paper.



Acknowledgements

We would like to thank the anonymous referees for the detailed and thoughtful comments and for the suggestions to improve the presentation.

REFERENCES

[1] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, and E. Zucca. An Effective Translation of $\mathcal{F}ickle_{\mathsf{II}}$ into Java. In A.Restivo, S.Ronchi, and L.Roversi, editors, *ICTCS'01*, volume 2002 of *LNCS*, pages 215–234. Springer, 2001.

[2] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, and E. Zucca. A Provenly Correct Translation of $\mathcal{F}ickle_{\mathsf{II}}$ into Java. Technical report, Part of Deliverable 4.2 of IST-2001-33477 (DART) project - http://www.cee.hw.ac.uk/DART/, 2003.

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

[4] K. Arnold, J. Gosling, and D. Holmes. *The Java™ Programming Language, Third Edition.* Addison-Wesley, 2000.

[5] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structual Operational Semantics of Multi-Threaded Java. In J. Alves-Foss, editor, *FSSOJ*, volume 1523 of *LNCS*, pages 157–200. Springer, 1999.

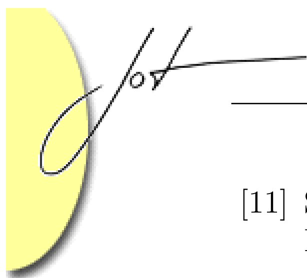
[6] C. Chambers. Predicate Classes. In O. Nierstrasz, editor, *ECOOP'93*, volume 707 of *LNCS*, pages 268–296. Springer, 1993.

[7] F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Re-classification and multi-threading: $\mathcal{F}ickle_{\mathsf{MT}}$. In H. Haddad, A. Omicini, R. L. Wainwright, and L. M. Liebrock, editors, *OOPS track at SAC'04*, volume 2, pages 1297–1304. ACM Press, 2004.

[8] F. Damiani, S. Drossopoulou, and P. Giannini. Refined Effects for Unanticipated Object Re-classification: $\mathcal{F}ickle3$ (extended abstract). In C. Blundo and C.Laneve, editors, *ICTCS'03*, volume 2841 of *LNCS*, pages 97–110. Springer, 2003.

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

[10] 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.



[11] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More Dynamic Object Re-classification: $\mathcal{F}ickle_{II}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

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

[13] G. Ghelli and D. Palmerini. Foundations of Extended Objets with Roles (extended abstract). In *FOOL'6*, 1999.

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

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

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

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

[18] K. Wight and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

ABOUT THE AUTHORS

Ferruccio Damiani[4] is researcher at the Dipartimento di Informatica, Università degli Studi di Torino, Italy. He can be reached at damiani@di.unito.it. His home page is http://www.di.unito.it/˜damiani.

Mariangiola Dezani-Ciancaglini[5] is full professor at the Dipartimento di Informatica, Università degli Studi di Torino, Italy. She can be reached at dezani@di.unito.it. Her home page is http://www.di.unito.it/˜dezani.

Paola Giannini[6] is associate professor at the Dipartimento di Informatica, Università del Piemonte Orientale, Italy. She can be reached at giannini@di.unito.it. Her home page is http://www.di.unito.it/˜giannini.

[4]Partially supported by EU within the FET - Global Computing initiative, project DART IST-2001-33477.

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

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