A state abstraction for coordination in JAVA-like languages*

Ferruccio Damiani[1], Elena Giachino[1], Paola Giannini[2], Nick Cameron[3], and Sophia Drossopoulou[3]

[1] Dipartimento di Informatica, Università di Torino
({damiani,giachino}@di.unito.it)
[2] Dipartimento di Informatica, Università del Piemonte Orientale
(giannini@mfn.unipm.it)
[3] Department of Computing, Imperial College
({N.Cameron,S.Drossopoulou}@imperial.ac.uk)

Abstract. Objects' *state*, intended as some abstraction over the value of fields, is always in the mind of a COOL (Concurrent Object-Oriented Language) programmer. In fact, as the state of an object changes so does its coordination behaviour.
We introduce a language feature for expressing the notion of state in JAVA-like languages. The proposed feature takes the form of *state class*, a new kind of class, equipped with a static type and effect system guaranteeing that during the execution of a method on a receiver `o`: (1) Even though the state of `o` may vary through states with different parameters, no attempt will be made to access non-existing parameters, and (2) No method invoked on a receiver different from `this` may cause (through method calls on `o`) a change in the state of `o`.

1 Introduction

The aim of this paper is to provide a simple, safe, and usable construct to make explicit the object's state abstraction that is always in the mind of a COOL (Concurrent Object-Oriented Language) programmer. The proposed construct takes the form of a new kind of class, that we call *state class*. State classes express methods' availability and behaviour in the form of a finite state automaton.

The state class construct is a *boundary coordination* mechanism, since it achieves *encapsulation by callee-side coordination* by design. "The idea of encapsulation by callee-side coordination is to implement concurrency coordination at the side of the callee, i.e., in the class that is accessed concurrently" [14] (Sect. 2.2). We refer to [14] (Sect. 4.2) for a survey of several COOLs with boundary coordination. In particular, the state class construct is related to the

* Work partially supported by MIUR PRIN'04 EOS project, and the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. The funding bodies are not responsible for any use that might be made of the results presented here.

actor model [1] and to the *behaviour abstraction* and *enable (a.k.a. behaviour) sets* proposals [10, 16]. These pioneering proposals have demonstrated the practical relevance of mechanisms to express object's state abstractions. However, they have not addressed the crucial issue of safety and the related issue of designing suitable type systems to provide compile time guarantees on the code. For instance, in the programming language ACT++ [10], enable sets are provided as a class library for concurrent programming in C++, so, the enable set mechanism is not part of the language. However, including features directly in the language (rather than indirectly as a library), has the advantage that it allows the programmer to write and think directly in terms of these features, thus writing better programs, and, as pointed out in [2], allows the compiler to produce better code and warn programmers of potential and actual problems.

Type systems for concurrent objects providing non-uniform services (that is, services that may vary according to the internal state of the objects) have been investigated both from a foundational perspective and from a programming perspective. From a foundational perspective, *regular object types* are a tool for describing objects interfaces that change along state transition [13] and the concurrent object calculus *TyCO* provides a formalization for the notion of "non-uniform" service availability [15]. From a programming perspective, F*ickle*'s *dynamic object re-classification* [6] (that is, changing the class membership of an object at run-time while retaining its identity) provides a state abstraction for JAVA-like languages by identifying (abstract) states with classes. F*ickle* focuses on a single-threaded setting, providing a type and effect system that is able to ensure (at compile-time) that no attempt is made to access non-existing members of an object, even though objects may be re-classified across classes with different members. F$ickle_{\mathsf{MT}}$ [5] adapts F*ickle*'s ideas to a multi-threaded setting, by proposing a compile-time type and effect analysis that introduces run-time checks. These checks delay the execution of threads that might cause access of non-existing members of shared objects. A major limitation of the F$ickle_{\mathsf{MT}}$ proposal [5] is that it is concentrated on the interaction between concurrency and dynamic object re-classification, while ignoring the general issue of synchronization control (which is supposed to be handled by means of other features of the language).

In this paper we take a programming perspective and propose a coordination mechanism for JAVA-like languages by relying on the notion of abstract state of an object. At the best of our knowledge, this is the first proposal of a construct for modelling a notion of object abstract state for expressing coordination in JAVA-like languages. Its main novelties are the ability of states to carry values (thanks to the presence of parameters) and the presence of a static type and effect system that, at compile time, is able to prevent *state parameter not found* errors and to enforce a *stronger notion of encapsulation by callee-side coordination* (see the discussion in Sect. 2).

This paper is organized as follows: Section 2 introduces and motivates the state class construct using an example. Section 3 gives syntax and typing rules, sketches operational semantics, and states type soundness of the FSJ calculus (a

```
public state class ReaderWriter {
  state FREE {
    public void shared() { this!!SHARED(1); }
    public void exclusive() { this!!EXCLUSIVE; }
  }
  state SHARED(int n)  {
    public void shared() { n++; }
    public void releaseShared() { n--; if (n==0) this!!FREE; }
  }
  state EXCLUSIVE {
    public void releaseExclusive() { this!!FREE; }
  }
}
```

Fig. 1. A multiple-reader, single-writer lock

minimal core calculus modelling the state class construct). Section 4 discusses some subtleties, limitations, and potentialities in the type and effect system. Section 5 briefly illustrates related work and Section 6 concludes by outlining some further work.

2 Overview and motivating example

In this section we introduce and motivate our proposal through an example written in a version of JAVA extended with state classes. The state class construct is designed to program objects that can be safely concurrently accessed. Therefore, in a state class, all the fields are `private` and all the methods are `synchronized` (that is, they are executed in mutual exclusion on the receiver object). A state class is allowed to extend an *ordinary* (i.e., non-state) class, but only state classes are allowed to extend state classes. Each state class specifies a collection of states. Each state has some *parameters* and declares some methods. The state of an object `o` can be changed only inside methods of `o`, by means of a *state transition statement*, `this!!`$\mathtt{S}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$, where "`S`" is the name of the target state and "$\mathtt{e}_1, \ldots, \mathtt{e}_n$" ($n \geq 0$) supply the values for all the parameters of `S`. An object belonging to a state class is always in one of the states specified in its class. Each state class constructor must set the state of the created object. The default constructor of the root of a hierarchy of state classes sets the state to the first state declared in the class.

The class `ReaderWriter` (in Fig. 1) implements a *multiple reader, single-writer lock* — see [3], for an implementation using traditional concurrency primitives in a dialect of MODULA 2, and [2], for an implementation using chords in POLYPHONIC C♯.

When a thread e invokes a method `m` on an object `o` belonging to a state class (e.g., to the class `ReaderWriter` in Fig. 1), if either `o` is in a state that does not support the invoked method (e.g., `shared` invoked on an `EXCLUSIVE ReaderWriter`) or some other thread is executing a method on `o`, then the execution of e is blocked until `o` reaches (because of the action of some other thread)

```
public state class ReaderWriterFair extends ReaderWriter {
  state SHARED(int n) {
    public void exclusive()
      { this!!PENDING_WRITER(n); pre_exclusive(); this!!EXCLUSIVE; }
  }
  state PENDING_WRITER(int n) {
    public void releaseShared() { n--; if (n==0) this!!PRE_EXCLUSIVE; }
  }
  state PRE_EXCLUSIVE {
    private void pre_exclusive() { }
  }
}
```

Fig. 2. A fair multiple-reader, single-writer lock

a `state` where the invoked method is available and no other thread is executing a method on `o`.

The policy implemented by the `ReaderWriter` class above is prone to writers' starvation. The class `ReaderWriterFair` (in Fig. 2) extends the class `ReaderWriter` to implement a writer starvation free policy.

An extending class inherits all the states of the extended class, and may add/override methods and introduce new states. Thus, class `ReaderWriterFair` has `state`s `FREE`, `SHARED`, `EXCLUSIVE`, `PENDING_WRITER` and `PRE_EXCLUSIVE`. When the request `exclusive()` is received by an object `o` in state `SHARED(`n`)`, then the `state` of `o` is set to `PENDING_WRITER(`n`)` and the method body suspends; in this state `o` can only execute up to n requests of `releaseShared()`; after the n-th such request, the state of `o` is set to `PRE_EXCLUSIVE`; in state `PRE_EXCLUSIVE` the method body for `exclusive` can continue, and will set the state of `o` to `EXCLUSIVE`.

The `ReaderWriterFair` class illustrates a common pattern in `state` class programming: the private method `pre_exclusive` has an empty body, and acts as a test that the receiver has reached the state `PRE_EXCLUSIVE`.

The state class construct is equipped with a static type and effect system guaranteeing that, during execution of a method on a state receiver `o`:

(1) No attempt will be made to access parameters not available in the current state of `o`, and
(2) No method invoked on a receiver (syntactically) different from `this` (or `super`)[4] may cause the invocation of a state method on `o` (and, therefore, cause a change of the state of `o` or an access to the state parameters of `o`).

Property (1) is, for state parameters, the standard requirement that well typed programs cannot cause a *field not found error*. To achieve this property the type and effect system estimates the *local (or intra-class) effect* of each expression occurring in the body of a method, that is, it traces how evaluating the expression

[4] For sake of readability, the `super` keyword is not included in the formal presentation of the state class construct given in Sect. 3.

may change the state of the receiver object. Typechecking of a method declaration in state `S` involves estimating the possible states of the receiver in the various points of the body of the method, by taking `S` as initial state of the receiver. Accessing an attribute is allowed only if the attribute is defined (with the same type) in all the (estimated) possible states of the receiver. For instance, in the body of method `releaseShared` in state `SHARED` of class `ReaderWriter` (see Fig. 1), both accesses to `n` are correct since the initial state in which the body is evaluated is `SHARED`. The local effect of the body is to possibly change the state of the receiver from `SHARED` to `FREE`. Note that, modifying class `ReaderWriter` by replacing method `releasedShared` in state `SHARED` with

```
public void releaseShared() { if (n==0) this!!FREE; n--; }
```

would cause a compile time type error since, the evaluation of the expression "`if (n==0) this!!FREE;`", may change the state of the receiver from `SHARED` to `FREE`. So availability of the parameter `n` for the expression "`n--`" is not guaranteed.

Property (2) strengthens the "standard" *encapsulation by callee-side coordination*. It prevents a family of programming errors that would not be prevented by the "standard" notion of encapsulation by callee-side coordination (given at the beginning of Sect. 1). For instance, modifying class `ReaderWriter` by replacing method `releaseShared` in state `SHARED` with

```
public void releaseShared(ReaderWriter x)
  { x.shared(); n--; if (n==0) this!!FREE; }
```

(note that `x` might be an alias of the current `this`) or with

```
public void releaseShared(C y) { y.m(); n--; if (n==0) this!!FREE; }
```

(taking a class `C` and a method `m()` such that the call `y.m()` may cause the invocation of a state method on the current `this` and, therefore, change its state or the value of the parameter `n`), may result (at run-time) in violations of the multiple-reader/single-writer protocol, deadlocks, or field not understood errors.

Neither example breaks the "standard" encapsulation by callee-side coordination, but both are rejected by our type and effect system. To achieve this behaviour, the type and effect system considers in addition to an expression's *local effect*, also its *global effect*, i.e., the potential changes of state of any objects in the heap (through method calls on the parameters).

The *global (or inter-class) effect* of an expression `e` is the set of the (state) classes of the objects that may be receiver of a state method invocation caused by the evaluation of `e`. We represent global effects as sets of (state or non-state) class names. Thus $\{\mathtt{A}_1, \ldots, \mathtt{A}_n\}$ denotes the set of all state classes `B` such that `B` is subclass of some $\mathtt{A}_i$ $(1 \leq i \leq n)$. For instance, {`Object`} represents the set of all the possible state classes.

3 FSJ: a core calculus for state classes

This section gives syntax, typing rules, operational semantics, and type soundness of FSJ, a minimal imperative core calculus for state classes. FSJ models the innovative features of the state class construct (namely state classes, state parameters and methods, `state` transitions, and the static type and effect system mentioned in Sect. 2) and multi-threaded computations. As we will briefly discuss in Sect. 4, the type and effect system could be straightforwardly extended to deal with standard language features, like fields, ordinary (that is, non-state) classes, interfaces, conditionals, and loops.

A FSJ program consists of a set of state class definitions plus an expression to be evaluated, that we will call the *main expression* of the program.

3.1 Syntax

The abstract syntax of FSJ class declarations (`L`), class constructor declarations (`K`), state declarations (`N`), method declarations (`M`), and expressions (`e`) is given in Fig. 3. The metavariables `A`, `B`, `C`, and `D` range over class names; `S` ranges over state names; `p` ranges over state parameter names; `m` ranges over method names; `x` ranges over method parameter names; and `a`, `b`, `c`, `d`, and `e` range over expressions.

We write "$\bar{\texttt{e}}$" as a shorthand for a possibly empty sequence "$\texttt{e}_1, \cdots, \texttt{e}_n$" (and similarly for `C`, `p`, `S`, `x`) and write "$\bar{\texttt{N}}$" as a shorthand for "$\texttt{N}_1 \cdots \texttt{N}_n$" with no commas (and similarly for $\bar{\texttt{M}}$). We write the empty sequence as "$\bullet$" and denote the concatenation of sequences using either comma or juxtaposition, as appropriate. We abbreviate operations on pair of sequences by writing "$\bar{\texttt{C}}\ \bar{\texttt{p}}$" for "$\texttt{C}_1\ \texttt{p}_1, \ldots, \texttt{C}_n\ \texttt{p}_n$", where n is the length of $\bar{\texttt{C}}$ and $\bar{\texttt{p}}$. We assume that sequences of state (parameter, or method) declarations do not contain duplicate names.

The class declaration `state class C extends D` $\{\texttt{K}\ \bar{\texttt{N}}\}$ defines a state class of name `C` with superclass `D`. The new class has a single constructor `K` and a set of states $\bar{\texttt{N}}$. The state declarations $\bar{\texttt{N}}$ may either refine (by adding/overriding methods) states that are already present in `D` or add new states.

The constructor declaration `C`$(\bar{\texttt{C}}\ \bar{\texttt{p}})$ $\{$`this!!S`$(\bar{\texttt{p}})\}$ specifies how to initialize the state and the state parameters of an instance of `C`. It takes exactly as many parameters as there are parameters of the state `S` and its body consists of a state transition statement.

The state declaration `state S`$(\bar{\texttt{C}}\ \bar{\texttt{p}})$ $\{\bar{\texttt{M}}\}$ introduces a state with name `S` and parameters of names $\bar{\texttt{p}}$ and types $\bar{\texttt{C}}$. The declaration provides a suite of methods $\bar{\texttt{M}}$ that are available in the state `S` of the class `C` containing the state declaration. A state `S` declared in a class `C` inherits all the (not overridden) methods that are defined in the (possible) declarations of `S` contained in the superclasses of `C`.

The method declaration `C m` $(\bar{\texttt{C}}\ \bar{\texttt{x}})$ $\texttt{D}_1, \ldots, \texttt{D}_n$ $\{\texttt{e}\}$, where $n \geq 0$, introduces a method named `m` with result type `C`, parameters $\bar{\texttt{x}}$ of types $\bar{\texttt{C}}$, global effect $\{\texttt{D}_1, \ldots, \texttt{D}_n\}$, and body `e`. The variables $\bar{\texttt{x}}$ and the pseudo-variable `this` are bound in `e`.

Syntax:

$$
\begin{array}{rcl}
\mathtt{L} & ::= & \mathtt{state\ class\ C\ extends\ C\ \{K\ \bar{N}\}} \\
\mathtt{K} & ::= & \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{p}})\{\mathtt{this!!S}(\bar{\mathtt{p}})\} \\
\mathtt{N} & ::= & \mathtt{state\ S}\ (\bar{\mathtt{C}}\ \bar{\mathtt{p}})\{\bar{\mathtt{M}}\} \\
\mathtt{M} & ::= & \mathtt{C\ m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \bar{\mathtt{C}}\ \{\mathtt{e}\} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{this} \mid \mathtt{this.p} \mid \mathtt{e;\ e} \mid \mathtt{new\ C}(\bar{\mathtt{e}}) \mid \mathtt{this!!S}(\bar{\mathtt{e}}) \mid \mathtt{spawn(e)} \mid \mathtt{e.m}(\bar{\mathtt{e}})
\end{array}
$$

Subtyping:

$$\mathtt{C} <: \mathtt{C} \qquad \frac{\mathtt{C}_1 <: \mathtt{C}_2 \qquad \mathtt{C}_2 <: \mathtt{C}_3}{\mathtt{C}_1 <: \mathtt{C}_3} \qquad \frac{\mathtt{state\ class\ C}_1\ \mathtt{extends\ C}_2\ \cdots\ \{\cdots\}}{\mathtt{C}_1 <: \mathtt{C}_2}$$

State name lookup:

$$states(\mathtt{Object}) = \emptyset \qquad \frac{\mathtt{state\ class\ C\ extends\ D}\ \{\mathtt{K\ state\ S}_1(\cdots)\{\cdots\}\cdots\mathtt{state\ S}_n(\cdots)\{\cdots\}\}}{states(\mathtt{C}) = \{\mathtt{S}_1, \ldots, \mathtt{S}_n\} \cup states(\mathtt{D})}$$

State parameters lookup:

$$\frac{\mathtt{state\ class\ C}\ \cdots\ \{\cdots\mathtt{state\ S}(\bar{\mathtt{C}}\ \bar{\mathtt{p}})\{\cdots\}\cdots\}}{parameters(\mathtt{C}, \mathtt{S}) = \bar{\mathtt{C}}\ \bar{\mathtt{p}}} \qquad \frac{\mathtt{state\ class\ C\ extends\ D}\ \{\mathtt{K}\ \bar{\mathtt{N}}\} \qquad \mathtt{S} \notin \bar{\mathtt{N}}}{parameters(\mathtt{C}, \mathtt{S}) = parameters(\mathtt{D}, \mathtt{S})}$$

Method definition, method definition type, and method definition global effect lookup:

$$\frac{\mathtt{state\ class\ C}\ \cdots\ \{\mathtt{K}\ \bar{\mathtt{N}}\} \qquad \mathtt{state\ S}\{\bar{\mathtt{M}}\} \in \bar{\mathtt{N}} \qquad \mathtt{A\ m}(\bar{\mathtt{A}}\ \bar{\mathtt{x}})\ \mathtt{C}_1, \ldots, \mathtt{C}_n\ \{\mathtt{e}\} \in \bar{\mathtt{M}}}{\begin{array}{l} mDef(\mathtt{m}, \mathtt{C}, \mathtt{S}) = \mathtt{A\ m}(\bar{\mathtt{A}}\ \bar{\mathtt{x}})\ \mathtt{C}_1, \ldots, \mathtt{C}_n\ \{\mathtt{e}\} \\ mDefT(\mathtt{m}, \mathtt{C}, \mathtt{S}) = \bar{\mathtt{A}} \to \mathtt{A} \\ mDefGE(\mathtt{m}, \mathtt{C}, \mathtt{S}) = \{\mathtt{C}_1, \ldots, \mathtt{C}_n\} \end{array}}$$

$$\frac{\mathtt{state\ class\ C\ extends\ D}\ \{\mathtt{K}\ \bar{\mathtt{N}}\} \qquad (\mathtt{S} \notin \bar{\mathtt{N}} \quad \text{or} \quad (\mathtt{state\ S}\{\bar{\mathtt{M}}\} \in \bar{\mathtt{N}} \quad \text{and} \quad \mathtt{m} \notin \bar{\mathtt{M}}))}{\begin{array}{l} mDef(\mathtt{m}, \mathtt{C}, \mathtt{S}) = mDef(\mathtt{m}, \mathtt{D}, \mathtt{S}) \\ mDefT(\mathtt{m}, \mathtt{C}, \mathtt{S}) = mDefT(\mathtt{m}, \mathtt{D}, \mathtt{S}) \\ mDefGE(\mathtt{m}, \mathtt{C}, \mathtt{S}) = mDefGE(\mathtt{m}, \mathtt{D}, \mathtt{S}) \end{array}}$$

Method type lookup and method global effect lookup:

$$\frac{\{\mathtt{S}_1, \ldots, \mathtt{S}_n\} = \{\mathtt{S} \mid mDef(\mathtt{m}, \mathtt{C}, \mathtt{S}) \text{ defined}\} \quad n \geq 1 \quad \bar{\mathtt{A}} \to \mathtt{A} = mDefT(\mathtt{m}, \mathtt{C}, \mathtt{S}_1) = \cdots = mDefT(\mathtt{m}, \mathtt{C}, \mathtt{S}_n)}{mT(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{A}} \to \mathtt{A}}$$

$$\frac{\{\mathtt{S}_1, \ldots, \mathtt{S}_n\} = \{\mathtt{S} \mid mDef(\mathtt{m}, \mathtt{C}, \mathtt{S}) \text{ defined}\} \quad n \geq 1 \quad \gamma = mDefGE(\mathtt{m}, \mathtt{C}, \mathtt{S}_1) \cup \cdots \cup mDefGE(\mathtt{m}, \mathtt{C}, \mathtt{S}_n)}{mGE(\mathtt{m}, \mathtt{C}) = \gamma}$$

Fig. 3. FSJ syntax, subtyping rules, and lookup functions

The class declarations in a program must satisfy the following conditions: (1) `Object` is a distinguished class name whose declaration does *not* appear in the program; (2) For every class name `C` (except `Object`) appearing anywhere in the program, one and only one class with name `C` is declared in the program; and (3) The subtype relation induced by the class declarations in the program (denoted by $<:$ and formally defined in Fig. 3) is acyclic. To simplify the notation in what follows (as in [8]), we always assume a *fixed* program.

The lookup functions are given in Fig. 3. We write $\mathtt{S} \notin \bar{\mathtt{N}}$ to mean that no declaration of the state $\mathtt{S}$ is included in $\bar{\mathtt{N}}$, and $\mathtt{m} \notin \bar{\mathtt{M}}$ to mean that no declaration of the method $\mathtt{m}$ is included in $\bar{\mathtt{M}}$.

Lookup of the state names of a class $\mathtt{C}$, written $\mathit{states}(\mathtt{C})$, returns the set of the names of all the states declared in $\mathtt{C}$ or in its superclasses. `Object`, which is the unique non-state class of the language,[5] has the empty set of states.

Lookup of the parameters of a state $\mathtt{S}$ of a class $\mathtt{C}$, written $\mathit{parameters}(\mathtt{C},\mathtt{S})$, returns a sequence $\bar{\mathtt{C}}\,\bar{\mathtt{p}}$ pairing the type of each parameter of the state with its name.

Lookup of the definition of the method $\mathtt{m}$ in the state $\mathtt{S}$ of a state class $\mathtt{C}$ is denoted by $\mathit{mDef}(\mathtt{m},\mathtt{C},\mathtt{S})$.[6] The type of a method is denoted by a pair, written $\bar{\mathtt{B}} \to \mathtt{B}$, of a sequence of argument types $\bar{\mathtt{B}}$ and a result type $\mathtt{B}$.

Lookup of the type and lookup of the global effect specified in the definition of the method $\mathtt{m}$ in the state $\mathtt{S}$ of a state class $\mathtt{C}$ are denoted by $\mathit{mDefT}(\mathtt{m},\mathtt{C},\mathtt{S})$ and $\mathit{mDefGE}(\mathtt{m},\mathtt{C},\mathtt{S})$, respectively. Lookup of the type of the method $\mathtt{m}$ in class $\mathtt{C}$, written $\mathit{mT}(\mathtt{m},\mathtt{C})$, returns the unique type specified in the definitions of $\mathtt{m}$ in all the states of $\mathtt{C}$ (the function is undefined if either there is no definition for $\mathtt{m}$ in all the state of $\mathtt{C}$ or the various definitions of $\mathtt{m}$ do no specify the same type).

Lookup of the global effect of the method $\mathtt{m}$ in class $\mathtt{C}$, written $\mathit{mGE}(\mathtt{m},\mathtt{C})$, returns the union of all the global effects specified in the various definitions of $\mathtt{m}$ in the states of $\mathtt{C}$ (the function is undefined if no definition for $\mathtt{m}$ is available).

Note that $\mathit{parameters}(\mathtt{C},\mathtt{S})$, $\mathit{mDef}(\mathtt{m},\mathtt{C},\mathtt{S})$, $\mathit{mDefT}(\mathtt{m},\mathtt{C},\mathtt{S})$, $\mathit{mDefGE}(\mathtt{m},\mathtt{C},\mathtt{S})$, $\mathit{mGE}(\mathtt{m},\mathtt{C})$, and $\mathit{mT}(\mathtt{m},\mathtt{C})$ are undefined when $\mathtt{C} = \mathtt{Object}$.[7]

3.2 Typing

3.2.1 Global Effects, Global Effect Entailment, Local Effects, and Local Effect Tables

A *global (or inter-class) effect* is a set of classes. The metavariables α, β, γ, δ, and η range over global effects. We say that "the expression $\mathtt{e}$ has global effect γ" to mean that "the evaluation of $\mathtt{e}$ may invoke a state method of an object belonging to a class $\mathtt{C}$ such that $\mathtt{C} <: \mathtt{D}$ for some $\mathtt{D} \in \gamma$".

The *global effect entailment* relation (denoted by $\prec:$ and formally defined in Fig. 4) is such that "$\alpha \prec: \beta$" implies "$\{\mathtt{C} \mid \mathtt{C}$ is a state class and $\mathtt{C} <: \mathtt{A}$ for some $\mathtt{A} \in \alpha\} \subseteq \{\mathtt{D} \mid \mathtt{D}$ is a state class and $\mathtt{D} <: \mathtt{B}$ for some $\mathtt{B} \in \beta\}$".

A *local (or intra-class) effect* is a triple $\sigma_1 \Rightarrow_{\mathtt{C}} \sigma_2$ where $\mathtt{C}$ is a class name and σ_1 and σ_2 (called the *source* and the *target* of the effect, respectively) are non-empty sets of state names included in $\mathit{states}(\mathtt{C})$. We say that "the expression $\mathtt{e}$ has local effect $\sigma_1 \Rightarrow_{\mathtt{C}} \sigma_2$" to mean that "if `this` is bound to an object $\mathtt{o}$ of class $\mathtt{C}$ that is in one of the states in σ_1, then the evaluation of $\mathtt{e}$ leaves $\mathtt{o}$ in one of the states in σ_2".

[5] In a real language other non-state classes may be declared.

[6] In a real language, the lookup functions should be suitably modified to take into account method overloading, that (for simplicity) is not included in FSJ.

[7] In a real language the class `Object` would have several methods.

Global efffect entailment:

$$\frac{\text{for all } \mathtt{C}_1 \in \gamma_1, \quad \text{for some } \mathtt{C}_2 \in \gamma_2, \quad \mathtt{C}_1 <: \mathtt{C}_2}{\gamma_1 \prec: \gamma_2}$$

Fig. 4. FSJ global effects entaiment

A *local effect table for a class* C is a mapping Θ from method-state pairs to sets of states σ, such that:

- $\mathrm{Dom}(\Theta) = \{(\mathtt{m}, \mathtt{S}) \mid \mathit{mDef}(\mathtt{m}, \mathtt{C}, \mathtt{S}) \text{ defined}\}$, and
- for all $(\mathtt{m}, \mathtt{S}) \in \mathrm{Dom}(\Theta)$, $\emptyset \neq \Theta(\mathtt{m}, \mathtt{S}) \subseteq \mathit{states}(\mathtt{C})$.

It can be seen as the assertion that, for all m and S such that method m is defined in state S of class C, the body of $\mathit{mDef}(\mathtt{m}, \mathtt{C}, \mathtt{S})$ has local effect $\{\mathtt{S}\} \Rightarrow_{\mathtt{C}} \Theta(\mathtt{m}, \mathtt{S})$. We will write "$\Theta$ FOR C" to mean that "Θ is a local effect table for C".

3.2.2 Typing expressions, method declarations, and class declarations

The typing rules for expressions, method declarations, and class declarations are given in Fig. 5. An *environment* Π is a mapping from method parameters to types, written $\bar{\mathtt{x}} : \bar{\mathtt{C}}$. The typing judgment of expressions has the form $\mathtt{D} \mid \Theta \mid \Pi \vdash_{\mathrm{exp}} \mathtt{e} : \mathtt{E} \mid \gamma \mid \sigma_1 \Rightarrow_{\mathtt{D}} \sigma_2$, where D is the class of this, Θ is a local effect table for D, and Π contains the type assumptions for the method parameters occurring in e. The judgement must be read "with this of class D, local effect table Θ, and environment Π, the expression e has type E, global effect γ, and local effect $\sigma_1 \Rightarrow_{\mathtt{D}} \sigma_2$". We will write $\bar{\mathtt{C}} <: \bar{\mathtt{D}}$ as shorthand for $\mathtt{C}_1 <: \mathtt{D}_1, \ldots, \mathtt{C}_n <: \mathtt{D}_n$.

The rules for expressions are syntax directed. Evaluating a variable (i.e., a state or method parameter) or the pseudo-variable this cannot invoke a state method on any object and cannot change the state of the current object, so the corresponding rules assign to these expressions the empty global effect and a local effect of the shape $\sigma \Rightarrow_{\mathtt{D}} \sigma$, where D is the class of this. Note that the rule for state parameter selection checks that the selected parameter is defined, with the same type, in all the states in σ.

The rule for sequential composition assigns to "$\mathtt{e}_1; \mathtt{e}_2$" the type of "$\mathtt{e}_2$" and propagates the global and the local effects of "$\mathtt{e}_1$" and "$\mathtt{e}_2$". Global effects are propagated by taking the set-theoretic union of the global effects of "$\mathtt{e}_1$" and "$\mathtt{e}_2$", while local effects are propagated by ensuring that the source of the effect of "$\mathtt{e}_2$" and the target of the effect of "$\mathtt{e}_1$" are equal and taking, as source, the source of the effect of "$\mathtt{e}_1$" and, as target, the target of the effect "$\mathtt{e}_2$", respectively.

The rules for object creation, state transition, and thread spawning are quite intuitive: they just perform fairly standard checks on types and propagate the effects of their subexpressions — the target of the effect assigned by the rule for state transition is the singleton set containing the target state of the transition.

Expression typing:

$$\mathtt{D} \mid \Theta \mid \Pi, \mathtt{x} : \mathtt{E} \vdash_{\text{exp}} \mathtt{x} : \mathtt{E} \mid \emptyset \mid \sigma \Rightarrow_{\mathtt{D}} \sigma \qquad \text{(T-Var1)}$$

$$\frac{\text{for all } \mathtt{S} \in \sigma, \quad \mathtt{E}\ \mathtt{p} \in \mathit{parameters}(\mathtt{D}, \mathtt{S})}{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{this.p} : \mathtt{E} \mid \emptyset \mid \sigma \Rightarrow_{\mathtt{D}} \sigma} \qquad \text{(T-Var2)}$$

$$\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{this} : \mathtt{D} \mid \emptyset \mid \sigma \Rightarrow_{\mathtt{D}} \sigma \qquad \text{(T-This)}$$

$$\frac{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_1 : \mathtt{E}_1 \mid \gamma_1 \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_1 \qquad \mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_2 : \mathtt{E}_2 \mid \gamma_2 \mid \sigma_1 \Rightarrow_{\mathtt{D}} \sigma_2}{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_1; \mathtt{e}_2 : \mathtt{E}_2 \mid \gamma_1 \cup \gamma_2 \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_2} \qquad \text{(T-Seq)}$$

$$\frac{\begin{array}{l}\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_1 : \mathtt{A}_1 \mid \gamma_1 \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_1 \quad \cdots \quad \mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_n : \mathtt{A}_n \mid \gamma_n \mid \sigma_{n-1} \Rightarrow_{\mathtt{D}} \sigma_n \\ \mathtt{state\ class\ C} \cdots \{\mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{p}})\{\cdots\} \cdots\} \qquad \mathtt{A}_1, \ldots, \mathtt{A}_n <: \bar{\mathtt{C}}\end{array}}{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{new\ C}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{C} \mid \gamma_1 \cup \cdots \cup \gamma_n \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_n} \qquad \text{(T-New)}$$

$$\frac{\begin{array}{l}\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_1 : \mathtt{A}_1 \mid \gamma_1 \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_1 \quad \cdots \quad \mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_n : \mathtt{A}_n \mid \gamma_n \mid \sigma_{n-1} \Rightarrow_{\mathtt{D}} \sigma_n \\ \mathit{parameters}(\mathtt{D}, \mathtt{S}) = \bar{\mathtt{C}}\ \bar{\mathtt{p}} \qquad \mathtt{A}_1, \ldots, \mathtt{A}_n <: \bar{\mathtt{C}}\end{array}}{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{this!!S}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{Object} \mid \gamma_1 \cup \cdots \cup \gamma_n \mid \sigma_0 \Rightarrow_{\mathtt{D}} \{\mathtt{S}\}} \qquad \text{(T-Trans)}$$

$$\frac{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e} : \mathtt{C} \mid \gamma \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_1 \qquad mT(\mathtt{run}, \mathtt{C}) = \bullet \rightarrow \mathtt{Object}}{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{spawn(e)} : \mathtt{Object} \mid \gamma \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_1} \qquad \text{(T-Spawn)}$$

$$\frac{\begin{array}{l}\mathtt{e} \neq \mathtt{this} \qquad \mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e} : \mathtt{A} \mid \gamma_0 \mid \sigma \Rightarrow_{\mathtt{D}} \sigma_0 \\ \mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_1 : \mathtt{A}_1 \mid \gamma_1 \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_1 \quad \cdots \quad \mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_n : \mathtt{A}_n \mid \gamma_n \mid \sigma_{n-1} \Rightarrow_{\mathtt{D}} \sigma_n \\ mT(\mathtt{m}, \mathtt{A}) = \bar{\mathtt{E}} \rightarrow \mathtt{E} \qquad \mathtt{A}_1, \ldots, \mathtt{A}_n <: \bar{\mathtt{E}} \qquad mGE(\mathtt{m}, \mathtt{A}) = \gamma \qquad \{\mathtt{D}\} \not<: \gamma\end{array}}{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e.m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{E} \mid \gamma \cup \gamma_0 \cup \cdots \cup \gamma_n \mid \sigma \Rightarrow_{\mathtt{D}} \sigma_n} \qquad \text{(T-NonThisInvk)}$$

$$\frac{\begin{array}{l}\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_1 : \mathtt{A}_1 \mid \gamma_1 \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma_1 \quad \cdots \quad \mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{e}_n : \mathtt{A}_n \mid \gamma_n \mid \sigma_{n-1} \Rightarrow_{\mathtt{D}} \sigma_n \\ mT(\mathtt{m}, \mathtt{D}) = \bar{\mathtt{E}} \rightarrow \mathtt{E} \qquad \mathtt{A}_1, \ldots, \mathtt{A}_n <: \bar{\mathtt{E}} \qquad mGE(\mathtt{m}, \mathtt{D}) = \gamma \\ \sigma' = \begin{cases} \sigma_n & \text{if } \sigma_n \subseteq \{\mathtt{S} \mid \mathit{mDef}(\mathtt{m}, \mathtt{D}, \mathtt{S}) \text{ defined}\} \\ \{\mathtt{S} \mid \mathit{mDef}(\mathtt{m}, \mathtt{D}, \mathtt{S}) \text{defined}\} & \text{otherwise} \end{cases} \\ \sigma'' = \cup_{\mathtt{S} \in \sigma'} \Theta(\mathtt{m}, \mathtt{S})\end{array}}{\mathtt{D} \mid \Theta \mid \Pi \vdash_{\text{exp}} \mathtt{this.m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{E} \mid \gamma \cup \gamma_1 \cup \cdots \cup \gamma_n \mid \sigma_0 \Rightarrow_{\mathtt{D}} \sigma''} \qquad \text{(T-ThisInvk)}$$

Method typing:

$$\frac{\mathtt{C} \mid \Theta \mid \bar{\mathtt{x}} : \bar{\mathtt{A}} \vdash_{\text{exp}} \mathtt{e} : \mathtt{B} \mid \gamma \mid \{\mathtt{S}\} \Rightarrow_{\mathtt{C}} \sigma \qquad \mathtt{B} <: \mathtt{A} \qquad \gamma \prec: \{\bar{\mathtt{E}}\} \qquad \Theta(\mathtt{m}, \mathtt{S}) = \sigma}{\mathtt{A\ m}\ (\bar{\mathtt{A}}\ \bar{\mathtt{x}})\ \bar{\mathtt{E}}\ \{\mathtt{e}\}\ \text{ OK IN S OF C WRT } \Theta} \qquad \text{(T-Method)}$$

Well formed class:

$$\frac{\begin{array}{l}\mathtt{K} = \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{p}})\{\mathtt{this!!S}_0(\bar{\mathtt{p}})\} \qquad \mathit{parameters}(\mathtt{C}, \mathtt{S}_0) = \bar{\mathtt{C}}\ \bar{\mathtt{p}} \\ \text{for all } \mathtt{state\ S}(\bar{\mathtt{D}}\ \bar{\mathtt{q}})\{\bar{\mathtt{M}}\} \in \bar{\mathtt{N}}, \\ \quad \text{if } \mathit{parameters}(\mathtt{D}, \mathtt{S}) \text{ defined then } \mathit{parameters}(\mathtt{D}, \mathtt{S}) = \bar{\mathtt{D}}\ \bar{\mathtt{q}} \\ \quad \text{for all } \mathtt{A\ m}\ (\bar{\mathtt{A}}\ \bar{\mathtt{x}})\ \bar{\mathtt{B}}\ \{\mathtt{e}\} \in \bar{\mathtt{M}}, \\ \qquad \{\mathtt{C}\} \prec: \{\bar{\mathtt{B}}\}, \quad mT(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{A}} \rightarrow \mathtt{A}, \quad \text{and} \\ \qquad \text{if } mT(\mathtt{m}, \mathtt{D}) \text{ defined, then } mT(\mathtt{m}, \mathtt{D}) = \bar{\mathtt{A}} \rightarrow \mathtt{A} \text{ and } \{\bar{\mathtt{B}}\} \prec: mGE(\mathtt{m}, \mathtt{D}) \\ \Theta \text{ FOR C} \qquad \text{for all } (\mathtt{m}, \mathtt{S}) \in \mathrm{Dom}(\Theta), \quad \mathit{mDef}(\mathtt{m}, \mathtt{C}, \mathtt{S}) \text{ OK IN S OF C WRT } \Theta\end{array}}{\mathtt{state\ class\ C\ extends\ D}\ \{\mathtt{K}\ \bar{\mathtt{N}}\}\ \text{ OK}} \qquad \text{(T-Class-Ok)}$$

Fig. 5. FSJ typing rules

Note that thread spawning and state transition have type `Object`; this ensures that convoluted expressions, like $\texttt{this!!S}_1(\texttt{p}_1).\texttt{p}$ and $\texttt{spawn}(\texttt{e}).\texttt{p}$, are ill typed.[8]

For method call there are two rules: one applies if the receiver is `this`, and the other applies if the receiver is not `this`. Both the rules require that the method called is defined in some state of the class of the receiver, and that the type of each actual parameter is a subtype of the type of the corresponding formal parameter. For calls $\texttt{e.m}(\texttt{e}_1, \ldots, \texttt{e}_n)$ with $\texttt{e} \neq \texttt{this}$, it is also required that the evaluation of the body of `m` may not invoke a state method of an object of the type `D` of the current `this` ($\{\texttt{D}\} \not:\!\!\rightarrow \gamma$, where $\gamma = mGE(\texttt{m}, \texttt{A})$ and `A` is the type of `e`). The local effect of the call has, as source, the source of the local effect of `e` (the first subexpression that is evaluated) and, as target, the target of the local effect of $\texttt{e}_n$ (the last subexpression that is evaluated before the call). The global effect is the union of the global effects of `e`, of the formal parameters $\texttt{e}_i$, and of the method `m` in the class `A` of `e` ($\gamma = mGE(\texttt{m}, \texttt{A})$, which is the union of the global effects specified in each method declaration that might be bound to `m` in the method invocation).

For calls $\texttt{this.m}(\texttt{e}_1, \ldots, \texttt{e}_n)$, the method body bound to `m` may change the state of `this`. The set of states `this` may be in after the call is estimated by using the local effect table associated with the class of `this`. There are two cases. First, the invoked method is present in any (estimated) possible state of the target of the local effect of $\texttt{e}_n$ (the last subexpression that is evaluated before the call). Second, it is not. In the first case, after the call `this` can be in any state that can be reached by any state of σ_n through execution of the method `m`. In the second case, we have to consider the fact that, when evaluating the call, if `this` is in a state in which `m` is not defined, the computation is suspended waiting for a another thread to cause a change to a state in which `m` is defined. Therefore, after the evaluation of the call `this` could be in any state that can be reached by a state in which `m` is defined through execution of the method `m`. The global effect of the call is as for the previous case considering that the evaluation of the expression `this` has empty global effects.

For a method declaration `M`, the judgement `M` OK IN `S` OF `C` WRT Θ is to be read as "method declaration `M` is ok in state `S` of class `C` with respect to the local effect table Θ". The associated rule ensures that the body of `M`:

- typechecks with `this` of type `C`, local effect table Θ, and environment mapping each parameter of the method to its declared type;
- has a type which is a subtype of the return type declared in `M`;
- has a global effect which entails the one declared in `M`; and
- has a local effect with source $\{\texttt{S}\}$ and target equal to the entry in the local effect table Θ for method `m` in state `S`.

The typing judgement for class declarations has the form `L` OK, read "class declaration `L` is ok". Let `C` be the name of the class declared by `L`. The associated rule ensures that:

[8] In a real language convoluted expressions would be ruled out by distinguishing between expressions and statements (having type `void`).

– the class constructor initializes the state to one of the states of the class C (i.e., declared in C or in one of its superclasses) by supplying appropriate values for the parameters;
– for each state S declared in C, if the state S is defined in the superclass D of C then it must have the same parameters and, for each method m declared in S,
  • C belongs to the global effect $\{\bar{\mathtt{B}}\}$ indicated in the declaration of m,[9]
  • if a declaration of a method of name m occurs in another state declared in C, then it must have the same type, and
  • if the method m is defined in the superclass D of C, then it must have the same type and its global effect must be entailed by the global effect specified in the declaration of m in the state S of C (this guarantees that the global effect of m in C entails the global effect of m in D); and
– there exists a local effect table Θ such that, for each method declaration M declared or inherited in any state of C, M OK IN S OF C WRT Θ holds.

Remark 1 (Algorithms for typechecking and local effect tables). The set of possible local effect tables for a state class C is finite. Therefore, the existence of a local effect table for which C is type correct, as well as well-formedness of programs, is decidable. Note that if C does not contain (possibly mutually) recursive methods, then there is at most one local effect table for which C is type correct.

A straightforward and efficient algorithm computes a local effect table for C by inspecting the body of all the methods defined (that is, declared or inherited) in C, and is complete if C does not contain recursive methods.

Although we believe that restricting, or even forbidding, recursion in the methods defined in state classes would not compromise the usefulness of the state class construct (see the discussion in Sect. 4.2), we are currently investigating more powerful and still efficient algorithms for dealing with state classes with (possibly mutually) recursive methods.

3.2.3 Typing main expressions

The typing rules for expressions (in Fig. 5) are designed to type the body of a method inside a class. In order to (re)use them to type the main expression of a program, that does not occur inside a class, we extend the syntax of effects by adding the *dummy local effect* $\emptyset \Rightarrow_{\mathtt{Main}} \emptyset$, where the distinguished name Main is different from all the class names. We will write $\vdash_{\text{main}} \mathtt{e} : \mathtt{E} \mid \gamma$, to be read "main expression e has type E and global effect γ", to mean that "e does not contain occurrences of this and $\mathtt{Main} \mid \emptyset \mid \emptyset \vdash_{\text{exp}} \mathtt{e} : \mathtt{E} \mid \gamma \mid \emptyset \Rightarrow_{\mathtt{Main}} \emptyset$ holds".

[9] In a real language, the name of the state class where the method declaration occurs, C, would be automatically included in global effect indicated in the method declaration, thus saving the programmer from listing it in the global effect declaration. For instance, we have adopted this convention in the examples in Sect. 2. So, all the method declarations in Fig. 1 implicitly indicate global effect {ReaderWriter} and those in Fig. 2 indicate global effect {ReaderWriterFair}. Note that, without this convention the classes ReaderWriter and ReaderWriterFair would not typecheck.

3.3 Reduction and Type Soundness

To model multi-threaded computations we consider pairs "$\bar{e}, \mathcal{H}$", called *configurations*, where $\bar{e}$ is a sequence of $n \geq 1$ *runtime expressions* and $\mathcal{H}$ is a *heap* mapping *addresses* to *objects*. We call $\bar{e}$ a "sequence of threads", since each element e_i of the sequence $\bar{e}$ represents a thread of computation. *Addresses*, ranged over by the metavariable ι, are the elements of the denumerable set $\mathbf{I}$. *Objects* are finite mappings associating: (1) the distinguished name "class" to a class name indicating the class of the object; (2) the distinguished name "state" to a state name indicating the state of the object; and (3) a mapping associating a finite number (possibly zero) of state parameter names to addresses. Objects will be denoted by $[\![\mathsf{class} : \mathtt{C},\ \mathsf{state} : \mathtt{S},\ \bar{\mathtt{p}} : \bar{\iota}]\!]$. *Heaps*, ranged over by the metavariable $\mathcal{H}$, are finite mappings from addresses to objects. The metavariables a, b, c, d, and e range over runtime expressions. We write $\bar{a}$ as a shorthand for a possibly empty sequence $a_1 \cdots a_n$ and $\dot{a}$ as a shorthand for a possibly empty sequence of length almost one.

The reduction relation has the form "$\bar{a}\, b_1\, \bar{c}, \mathcal{H}_1 \longrightarrow \bar{a}\, b_2\, \bar{c}\, \dot{d}, \mathcal{H}_2$", read "configuration $\bar{a}\, b_1\, \bar{c}, \mathcal{H}_1$ reduces to configuration $\bar{a}\, b_2\, \bar{c}\, \dot{d}, \mathcal{H}_2$ in one step". The (empty or singleton) sequence $\dot{d}$ indicates that a new thread might have been spawned because of the reduction of a `spawn` expression. We write $\longrightarrow^\star$ for the reflexive and transitive closure of $\longrightarrow$. The reduction rules ensure, by using the standard notions of *evaluation context* and *redex*, that inside each thread the computation follows a call-by-value reduction strategy.

We say that, in a configuration "$e_1, \cdots, e_n, \mathcal{H}$" ($n \geq 1$), the thread e_i ($1 \leq i \leq n$) is: *completed*, if $e_i = \iota$ for some address ι; *suspended*, if e_i is such that its current redex is the invocation a method `m` on an object ι, where the method `m` is defined in *some* state of the class of ι, and either `m` is not available in the *current* state of ι, or the lock on ι is held by some other thread of the configuration.

A normal form containing only completed threads and *some* suspended thread models a *deadlocked* computation. We can now state the soundness result.

Theorem 2 (FSJ Type Soundness). *Let* $\vdash_{\mathrm{main}} \mathtt{e} : \mathtt{E} \mid \eta$ *and* $\mathtt{e}, \emptyset \longrightarrow^\star e_0 e_1 \cdots e_n, \mathcal{H}$ ($n \geq 0$)*, with* $e_0 e_1 \cdots e_n, \mathcal{H}$ *a normal form. Then: (1) the heap* $\mathcal{H}$ *is* well formed*; (2) the thread sequence* $e_0 e_1 \cdots e_n$ *is typable; (3) the type of* e_0 *is a subtype of the type of* `e` *and the global effect of* e_0 *entails the global effect of* `e`*; and (4) each* e_i *is either completed or suspended.*

4 Discussion

This section briefly discusses some subtleties, limitations, and potentialities in the type and effect system.

4.1 Subtleties

The typechecking of state classes involves some subtle points.

(S1) According to rule (T-Class-Ok), typechecking a state class requires renewed typechecking of all inherited methods.

(S2) According to rule(T-Class-Ok), the global effect of method declarations occurring in a state class `C` includes the class `C` itself.

(S3) According to rule (T-NonThisInvk), the body of a method declared or inherited in a state class `D` cannot invoke, on expressions syntactically different from `this`, a method containing (a superclass of) `D` in its global effect.

(S4) As pointed out in Remark 1 at the end of Sect. 3.2.2, we don't know whether there exists an efficient *complete* algorithm for computing local effect tables in the presence of recursive methods. The lack of such an algorithm would restrict the use of recursive methods in state classes in any feasible implementation.

Note that the above points are confined to the typechecking of state classes. Namely, if we extend the language with ordinary (that is, non-state) classes, then, for any such ordinary class `C`: (1) Typecheking `C` would not require renewed typechecking of inherited methods; (2) The global effect of a method declaration occurring in `C` need not include the class `C`; (3) The body of a method declared in `C` can invoke methods on expressions different from `this`, regardless of the relation between the global effect `{C}` and the global effect of the invoked method; and (4) Typechecking `C` would not require the computation of a local effect table; therefore there would be no restrictions on recursive methods in ordinary classes.

4.2 Limitations

Indeed, subtleties (S3) and (S4) in Sect. 4.1 point out limitations of the type and effect system.

(L1) It would be safe to relax rule (T-NonThisInvk) and subtlety (S3) so as to allow the body of a method declared or inherited in a state class `D` to invoke a method containing `D` in its global effect, provided that the receiver of that method call was guaranteed not to be an alias of `this`.
This relaxation would require some refinements of the type system which we plan to do in further work.

(L2) According to subtlety (S4), a feasible implementation may restrict the use of recursive methods in state classes.

However, we do not consider these limitations to be severe: Firstly, we are confident that these limitations do not affect the applicability of state classes in practical situations. Namely, we considered a sequence of case studies solving classical concurrent programming problem examples, including the "Santa Claus Problem" [4], and found that these limitations did not restrict the state class programming style. Secondly, we are confident that the limitations can be relaxed, and plan to study such relaxations in our further work.

4.3 Potentialities

The extension of the type and effect system to deal with

(P1) standard features, like fields, ordinary classes (remember the discussion in Sect. 4.1), interfaces (just denote global effects by sets of classes and interfaces, and let each method signature specified in an interface `I` to include `I` in its global effect declaration), conditionals, and loops, and
(P2) features useful for writing more compact code, like, for instance, the construct `this!!previous` that expresses the transition to previous state, or the construct `as` that specifies that the declaration of a method in a state is "as" in other state, (thus saving the programmer from duplicating the same method body in different states), and top level (that is, outside of any state) method declarations in state classes

seems quite straightforward.

5 Related Work

The literature related to the present paper was partially quoted in the introduction. Here we briefly discuss the relations with other recently proposed coordination mechanisms for JAVA-like languages. None of these proposals provides a construct for directly supporting the notion of abstract state of an object.

The languages JOIN JAVA [9] and POLYPHONIC C♯ [2] are based on an adaptation of the *Join Calculus* [7]. The synchronization mechanism of these languages relies on the *join pattern*, called *chord* in POLYPHONIC C♯, construct. Chords can be used to codify the state of an object through the pattern (illustrated, for instance, in [2]) of using private asynchronous method to carry object state. However, as pointed out by an anonymous referee of an earlier version of the present paper, this pattern "is potentially error prone, since one may fail to preserve the linearity appropriately, either by forgetting to call methods (leading to deadlock) or by calling too many (leading to multiple overlapping states)". With the state class construct the notion of object state becomes part of the language definition, thus eliminating the possibility of such errors.

In the JEEG programming language [12] synchronization conditions on an object `o` are expressed with *linear temporal logic constraints* involving the value of fields and the method invocation history of `o`. These constraints can be used to codify the state of an object `o`. However, state attributes have to be mapped on object fields and there is no way to express the fact that some fields should be accessible only in some states.

The JR programming language [11] extends JAVA providing a rich concurrency model with a variety of mechanisms for writing parallel and distributed programs. Besides the fact that none of these mechanisms directly support the notion of object state, it is worth noticing that the JR approach is quite different from the approach (followed in the present paper) of focusing on a single mechanism (namely the state class construct). Therefore, comparing state classes (as

well as JEEG temporal logic constraints or POLYPHONIC C♯ chords) with the JR proposal would indeed require investigating how state classes (resp. temporal logic constraints or chords) mix with the various features provided by JR.

6 Conclusions and Further Work

An extension of JAVA 1.4 with state classes *with parameter-free states and without effects*,[10] implemented through a preprocessor that produces plain JAVA code, is available at http://www.di.unito.it/~giannini/stateJimpl/.

Future work includes the integration of state parameters and effects, the exploration of alternative translation mechanisms, the development of benchmarks, and further investigations of the expressivity of the state class construct and on its integration in a full language.

Acknowledgements We thank Viviana Bono, Mario Coppo, Mariangiola Dezani-Ciancaglini, Jeremy Sproston, the referees of earlier versions of this paper, and the anonymous FTfJP'06 referees, for insightful comments and pointers to related work.

References

1. G. A. Agha. *ACTORS: A Model of Concurrency Computation in Distribuited Systems*. MIT Press, 1986.
2. N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C♯. *ACM TOPLAS*, 26(5):769–804, 2004.
3. A. D. Birrel. An introduction to programming with threads. Technical Report 35, DEC SRC, January 1989.
4. N. Cameron, F. Damiani, S. Drossopoulou, E. Giachino, and P. Giannini. Solving the Santa Claus problem using state classes. Technical report, Dip. di inf., Univ. di Torino, March 2006. Availble at http://www.di.unito.it/~damiani/papers/scp.pdf.
5. F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. On re-classification and multithreading. *JOT (www.jot.fm)*, 3(11):5–30, 2004. Special issue: OOPS track at SAC 2004.
6. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: $\text{Fickle}_{\text{ii}}$. *TOPLAS*, 24(2):153–191, 2002.
7. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join calculus. In *POPL'96*, pages 372–385. ACM, 1996.
8. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
9. G. S. Itzstein and D. Kearney. Join Java: an alternative concurrency semantics for Java. Technical Report ACRC-01-001, Univ. of South Australia, 2001.

[10] This unsafe variant of the state class construct does not affect the expressive power of the language, since state parameters can be straightforwardly codified by class fields.

10. D. G. Kafura and R. G. Lavender. Concurrent object-oriented languages and the inheritance anomaly. In T. Casavant, P Tvrdil, and F. Plásil, editors, *Parallel Computers: Theory and Practice*, pages 221–264. IEEE Press, 1996.
11. A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended java. *TOPLAS*, 26(3):578–608, 2004.
12. G. Milicia and V. Sassone. Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects. *Concurrency Computat.: Pract. Exper.*, 17(5-6):539–572, 2005.
13. O. Nierstrasz. Regular Types for Active Objects. In *OOPSLA'93*, volume 28 of *ACM SIGPLAN Notices*, pages 1–15, 1993.
14. M. Philippsen. A Survey of Concurrent Object-Oriented Languages. *Concurrency Computat.: Pract. Exper.*, 12(10):917–980, 2000.
15. A. Ravara and V. T. Vasconcelos. Typing Non-uniform Concurrent Objects. In *CONCUR'00*, volume 1877 of *LNCS*, pages 474–488, Berlin, 2000. Springer.
16. C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA'89*, pages 103–112. ACM, 1989.