Coalgebras and Monads in the Semantics of Java ⋆

Bart Jacobs and Erik Poll

Dept. of Computer Science, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.

Abstract

This paper describes the basic structures in the denotational and axiomatic semantics of sequential Java, both from a monadic and a coalgebraic perspective. This semantics is an abstraction of the one used for the verification of (sequential) Java programs using proof tools in the LOOP project at the University of Nijmegen. It is shown how the monadic perspective gives rise to the relevant computational structure in Java (composition, extension and repetition), and how the coalgebraic perspective offers an associated program logic (with invariants, bisimulations, and Hoare logics) for reasoning about the computational structure provided by the monad.

Key words: coalgebra, monad, Java

1 Introduction

This paper investigates the semantics of sequential Java from a combined coalgebraic/monadic perspective. It turns out that these separate perspectives are closely related—and that this is a more general phenomenon in the semantics of programming languages. Statements (and also expressions) in Java may have different termination options: they can hang (non-termination), terminate normally yielding a successor state (and possibly a result value), or terminate abruptly (for example, because of an exception). These three termination options can be captured by a suitable coproduct (disjoint union, or variant)

⋆ Extended version of [22].

Email address: {bart,erikpoll}@cs.kun.nl (Bart Jacobs and Erik Poll).
URL: http://www.cs.kun.nl/~{bart,erikpoll} (Bart Jacobs and Erik Poll).

type, describing the result of a computation. The semantics of a statement then takes the form,

$$S \times A \xrightarrow{\mathsf{stat}} \mathsf{Hang} + \mathsf{Normal} + \mathsf{Abrupt}$$

where S is the state space (set of states, or store) and A is the set of inputs (parameters). Java statements do not have parameters, but since Java methods do, it is convenient to also consider statements with parameters. It has become standard in semantics to describe the result type on the right hand side as a functor, acting on the state space S and the possible result value. In this setting we can work within the category **Sets** of sets and functions—and do not need domains—because the order structure used in denotational semantics comes from a standard (flat) order on the result type of statements, see Section 4.2.

Thus, the situation for Java involves a suitable functor $F\colon \mathbf{Sets} \times \mathbf{Sets} \to \mathbf{Sets}$ in two variables, describing the result type. Actually, this functor F is such that for a fixed set S of states, the functor $B \mapsto F(S,B)^S$ is a monad. Statements, acting on S with input A and output B, are then functions of the form

$$S \times A \xrightarrow{\mathsf{stat}} F(S,B).$$

An elementary but crucial observation is that there are two alternative (but equivalent) descriptions of such statements, namely as:

$$\textbf{Coalgebras} \quad S \longrightarrow F(S,B)^A \quad \text{of the functor } S \mapsto F(S,B)^A$$

or as:

$$\textbf{Kleisli maps} \quad A \longrightarrow F(S,B)^S \quad \text{of the monad } B \mapsto F(S,B)^S.$$

The equivalence relies on the following bijective correspondences obtained by Currying.

$$\frac{\dfrac{S \longrightarrow F(S,B)^A}{S \times A \longrightarrow F(S,B)}}{A \longrightarrow F(S,B)^S}$$

These two ways of viewing F as a functor in one parameter give two different dimensions to the semantics [1].

The main point of this paper is that these correspondences can be exploited fruitfully. For Java the monadic view yields a mathematically clean description of the basic underlying computational structure for the language, incorporated in its composition, extension and repetition (while, for, recursion)

[1] One can go further and distinguish a third parameter in the functor, for input (our A), as in [28]. But such extra generality is not relevant for our work.

constructs. This shows the appropriateness of computational monads (introduced in [37,38]) for an actual, real programming language. The coalgebraic view yields an associated program logic. It involves tailor made definitions of invariance, bisimulation and modal operators. The latter can be used to define an appropriate Hoare logic or dynamic logic (see *e.g.* [10]), providing reasoning principles for the computational structure obtained by the monad. For example, these principles can take the form of rules like $[s_1\,;s_2]\varphi \leftrightarrow [s_2][s_1]\varphi$, expressing that a formula φ holds after a composite statement $s_1\,;s_2$ if and only if $[s_1]\varphi$ holds after statement s_2, *i.e.* φ holds after first statement s_1 and then s_2.

The starting point for this work is the denotational semantics of sequential Java, that has been developed as part of the LOOP project (Logic of Object-Oriented Programming) [25,12,44]. This Java semantics provides the basis for formal reasoning about Java programs using theorem provers. A compiler has been developed, called the LOOP tool, which, given a sequential Java program, generates its semantics in a form that can serve as input to a theorem prover, see [5]. The theorem provers currently supported are PVS [40] and Isabelle [41], so the LOOP tool can generate the semantics of a Java program in several PVS or Isabelle theories. One of the aims of the LOOP project is to reason about a real programming language, warts and all; the Java semantics therefore covers essentially all of sequential Java, including details such as exceptions, breaks, and non-termination—but without inner classes and interactive I/O [2]. The semantics developed in the LOOP project has been used to reason about existing Java programs, for instance to prove a non-trivial safety property for the Vector class in the Java standard library [15]. Also, the LOOP tool works on a suitable annotation language for Java, called JML [31,32], providing for example class invariants and method specifications (with pre- and post-conditions). As a serious case study JML is being used for specifying and verifying the JavaCard API, see [42,43,6], used on the newest generation of Java-programmable smart cards. JML is briefly discussed in Subsection 7.2 below.

Here we will not describe the denotational semantics of all of the Java constructs, but concentrate on the use of a monad to (re)organise the semantics from a single perspective. A denotational semantics of Java is more complicated than the semantics typically considered in textbooks on denotational semantics. Not only does it involve the possibility of non-termination (using the familiar $\perp$), but it also involves different forms of abrupt termination

[2] In fact, strictly speaking I/O is not part of the Java language, but provided by classes in the Java API (Application Programming Interface). One way to accommodate I/O would be to include a model of the relevant API classes as part of the global state space, and to provide interpretations of the associated class methods for performing I/O.

of programs, such as exceptions and the different ways of "jumping" out of methods and repetitions via `break`, `return`, and `continue` statements. In our monad for Java we shall abstract from these different abrupt termination options, and consider only exceptions. The axiomatic semantics of exceptions is studied in for example [7,34,33]—mostly via a weakest precondition calculus—using a *single* possible exception, and not *many* forms of abrupt termination, like in Java. Here we show that the computational monad approach [37,38] provides a useful level of abstraction and a good means for organising all the complications that come with defining the semantics of a real programming language such as Java. The paper also provides a *post hoc* justification of the Java semantics as used in the LOOP project, by giving some of the central properties of the monadic structure and of the interpretation of some particular Java constructs. Not all aspects of the Java semantics are described in this paper; the representation of the global state space, used for relating identifiers to values, is discussed in [4], the treatment of inheritance in [13], and of exceptions in [20]. See also [12] for an overview.

The Java semantics used in the LOOP project has originally been developed primarily from a coalgebraic perspective [45,16,24]. This perspective focuses on the state space as a black box, and leads to useful notions such as class invariant, bisimilarity and modal logic. Hoare logics for Java and for JML have been developed "directly", see [14,12,23], but the modal operators associated with coalgebras (see [39,46,18,19]) also provide a *post hoc* justification in this case.

This paper is organised as follows. It starts in Section 3 with a sketch of the Java semantics as used in the LOOP project, focusing on the different abnormalities in Java. In the monadic approach in Section 4 these abnormalities are simplified to a single set E. This leads to a monad J, which we call the Java monad. Its Kleisli composition corresponds to Java composition, and its extension to Java extension. Furthermore, the homsets in its Kleisli category have a cppo structure. Section 5 considers this same situation for some other functors. Next, in Section 6, while statements and recursive statements are studied in this general framework. The cppo structure of Kleisli homsets also allows us to deal with recursive statements in the usual way. In the special case of our Java monad J, these general, denotational definitions are shown to be equivalent to more "operational" characterisations used in the LOOP project. The final section 7 describes modal operators associated with the functor capturing Java statements as coalgebras. These can be used for an appropriate axiomatic semantics for Java (and JML), taking the various termination options into account.

2 Preliminaries

We shall make frequent use of n-ary products $X_1 \times \cdots \times X_n$ of sets X_i, with projection functions $\pi_i \colon X_1 \times \cdots \times X_n \to X_i$. The empty product, when $n = 0$, describes a singleton set, which is written as $1 = \{*\}$. We also use n-ary coproducts (or disjoint unions) $X_1 + \cdots + X_n$ with coprojection (or injections) $\kappa_i \colon X_i \to X_1 + \cdots + X_n$. There is an associated "case" or "pattern match" construction which is perhaps not so familiar: given n functions $f_i \colon X_i \to Y$, there is a unique function[3] $f \colon X_1 + \cdots + X_n \to Y$ with $f \circ \kappa_i = f_i$, for all $1 \leq i \leq n$. We shall write

$$\begin{aligned} f(z) = \ & \mathsf{CASES}\ z\ \mathsf{OF}\ \{ \\ & \kappa_1(x_1) \mapsto f_1(x_1), \\ & \quad \vdots \\ & \kappa_n(x_n) \mapsto f_n(x_n)\ \} \end{aligned}$$

for the function that maps $z \in X_1 + \cdots + X_n$ of the form $\kappa_i(x_i)$ to $f_i(x_i)$.

Familiarity is assumed with the basics of the theory of monads, see for instance [38,49,26] for an introduction, or [30,36,3] for more advanced information.

3 Java semantics for verification

This section explains the essentials of the semantics of (sequential) Java as used in the LOOP project. As such it exists in the form of PVS and Isabelle/HOL definitions in higher order logic, in so-called prelude files, which form the basis for every verification exercise. Here we shall use a more mathematical notation for the basic ingredients of this semantics. Later in this paper it will be reformulated (and simplified) using a monad.

Traditionally, in denotational semantics an imperative program s is interpreted as a partial function on some state space S, *i.e.*

$$S \xrightarrow{[\![s]\!]} 1 + S.$$

The state space S is a global store giving the values of all program variables. We will not go into the precise form of the store here; for more detail, see [4].

[3] which, in categorical notation, is written as the cotuple $[f_1, \ldots, f_n]$.

The global state space S includes both the heap for allocating objects and their fields, and the stack for current bindings of identifiers, so a separate "environment" for bindings of identifiers as traditionally used in denotational semantics is not needed.

Above we have used the notation introduced in the previous section; the conventional notation for $1+S$ is $S_\perp$. The $1+\ldots$ option in the result type signals non-termination (or "hanging").

Similar to program statements, an expression e—possibly having side effects—is interpreted as a function

$$S \xrightarrow{[\![e]\!]} 1 + (S \times B).$$

Again, the first $+$-option 1 in the result type signals non-termination. The second option is for normal termination, which for expressions (of type B) yields a state, needed for side-effects, and a result value in B.

In a real programming language like Java however, things are more complicated. Statements and expressions in Java can not just hang or terminate normally, they can also terminate abruptly. Expressions can only terminate abruptly because of an exception (*e.g.* through division by 0), but statements may also terminate abruptly because of a `return` (to exit from a method call), `break` (to exit from a block, repetition or switch-statement), or `continue` (to skip the remainder of a repetition). The last two options can occur with or without label. Consequently, the result types of statements and expressions will have to be more complicated than the $1 + S$ and $1 + (S \times B)$ above. The result types of statements and expressions are abbreviated as $\mathsf{StatResult}(S)$ and $\mathsf{ExprResult}(S, B)$, where:

$$\mathsf{StatResult}(S) = 1 + S + \mathsf{StatAbn}(S)$$
$$\mathsf{ExprResult}(S, B) = 1 + (S \times B) + \mathsf{ExprAbn}(S)$$

Here $\mathsf{StatAbn}(S)$ and $\mathsf{ExprAbn}(S)$ are the types of statement and expression abnormalities, defined below.

Later in the monadic description we shall abstract away from the particular shapes of these abnormalities, but now we want to show what really happens in the semantics of Java (that is used for verification). Therefore, we describe all these abnormality options in appropriate definitions, involving a state space S. First, abnormal termination for statements is captured via four options:

$$\mathsf{StatAbn}(S) = (S \times \mathsf{RefType}) + S + (S \times (1 + \mathsf{String})) + (S \times (1 + \mathsf{String}))$$

where RefType and String are constant sets used for references and strings. The first +-option $S \times \mathsf{RefType}$ describes an exception result, consisting of a state and a reference to an exception. The second +-option is for a return result, the third one for a break result (possibly with a string as label), and the fourth one for a continue result (also possibly with a string as label).

Since exceptions are the only abnormalities that can result from expressions we have:

$$\mathsf{ExprAbn}(S) = S \times \mathsf{RefType}.$$

A void method `void m(A1 a1, ..., An an){ ... }` in Java is then interpreted as a state transformer function $S \times A1 \times \cdots \times An \to \mathsf{StatResult}(S)$. A non-void method `B m(A1 a1, ..., An an){ ... }` gets interpreted as a function $S \times A1 \times \cdots \times An \to \mathsf{ExprResult}(S, B)$. Notice that these state transformers can be described as coalgebras, namely of the functors $S \mapsto \mathsf{StatResult}(S)^{A1 \times \cdots \times An}$ and $S \mapsto \mathsf{ExprResult}(S, B)^{A1 \times \cdots \times An}$. Statements inside a method body are also translated as state transformers $S \to \mathsf{StatResult}(S)$, without parameters, of course. They are composed via an explicit composition operation, which we describe in the next paragraph. A whole class is represented as a single coalgebra, combining (via tupling) separate operations for all the fields, methods and constructors of the class. The class's fields are related to appropriate memory positions via a predicate on this coalgebra [4]. Similarly, the class's methods and constructors are related to their implementations, see [12] for details. Reasoning about a particular class/coalgebra proceeds under the assumption that these "implementation" predicates hold for the coalgebra.

On the basis of this representation of statements and expressions all language constructs of (sequential) Java are translated into the (higher order) logics of PVS and Isabelle [25,14,13,4,44]. For instance, the composition $(s\,;t)\colon S \to \mathsf{StatResult}(S)$ of two statements $s, t\colon S \to \mathsf{StatResult}(S)$ is defined as:

$$\begin{aligned}(s\,;t) = \lambda x \in S.\ \mathsf{CASES}\ s(x)\ \mathsf{OF}\ \{ & \\ & \kappa_1(*) \mapsto \kappa_1(*), \\ & \kappa_2(x') \mapsto t(x'), \\ & \kappa_3(w) \mapsto \kappa_3(w)\,\}.\end{aligned} \tag{1}$$

This means that if $s(x)$ hangs or terminates abruptly, then $(s\,;t)(x) = s(x)$ so that t is not executed at all, and if $s(x)$ terminates normally resulting in a successor state x', then $(s\,;t)(x) = t(x')$ and t is executed on this successor state. Notice how abnormalities are propagated.

[4] The LOOP compiler computes these positions.

The Java evaluation strategy prescribes that arguments should be evaluated first, from left to right (see [11, §§ 15.7.4]). But so far we have used values as arguments, and not expressions possibly having side-effects. Restricting to the case with one argument, this means that for a statement $t\colon S \times A \to \mathsf{StatResult}(S)$ we still have to define an extension[5] t^* of t with type $t^*\colon (S \times (S \to \mathsf{ExprResult}(S, A))) \to \mathsf{StatResult}(S)$, namely as:

$$\begin{aligned} t^*(x, e) = \ & \mathsf{CASES}\ e(x)\ \mathsf{OF}\ \{ \\ & \kappa_1(*) \mapsto \kappa_1(*), \\ & \kappa_2(x', a) \mapsto t(x', a), \\ & \kappa_3(w) \mapsto \kappa_3(w)\ \} \end{aligned} \tag{2}$$

(and similarly for an expression instead of a statement t).

In the next section we shall see how composition and extension can be obtained from an underlying monad.

4 The monad for Java semantics and its properties

This section introduces an appropriate monad J for Java statements and expressions. Its (categorical) properties are investigated in some detail, with emphasis on extension and composition, and on the order on homsets of the Kleisli category $\mathbf{Kl}(J)$ of the "Java" monad J.

The first step is to simplify the situation from the previous section. This is done by ignoring the complicated structure of Java abnormalities, and using one fixed set E in place of both $\mathsf{StatAbn}$ and $\mathsf{ExprAbn}$. Then we can see a statement as a special form of expression, namely one with result type 1. Thus, our general state transformer functions are of the form:

$$S \times A \longrightarrow 1 + (S \times B) + (S \times E).$$

They can be described as maps $S \times A \to F(S, B)$, for a functor $F\colon \mathbf{Sets} \times \mathbf{Sets} \to \mathbf{Sets}$ as in the introduction, with, obviously, $F(S, B) = 1 + (S \times B) \times (S \times E)$. Within the LOOP semantics they are regarded as coalgebras:

$$S \longrightarrow \Big(1 + (S \times B) + (S \times E)\Big)^A.$$

[5] In PVS and Isabelle we use overloading and also write t for t^*.

But here we shall first look at them as morphisms

$$A \longrightarrow \Big(1 + (S \times B) + (S \times E)\Big)^S$$

in the Kleisli category of a monad. These two representations are of course equivalent (via Currying), but they give different perspectives. In the coalgebraic view the state space S plays a central role, but in the monadic view S is just one of the ingredients of the monad, like partiality and exceptions; and the emphasis is more functional and lies on input and output.

Definition 1 *Let* ***Sets*** *be the category of sets and functions. Fix two sets E for exceptions and S for states. A functor $J\colon$* ***Sets*** $\to$ ***Sets*** *is defined by*

$$J(A) = (1 + (S \times A) + (S \times E))^S. \tag{3}$$

It forms a monad with unit and multiplication natural transformations:

$$A \xrightarrow{\eta_A} J(A) \qquad\qquad J^2(A) \xrightarrow{\mu_A} J(A)$$

given by

$$\eta_A(a) = \lambda x \in S.\, \kappa_2(x, a) \qquad \mu_A(f) = \lambda x \in S.\ \mathsf{CASES}\ f(x)\ \mathsf{OF}\ \{$$
$$\kappa_1(*) \mapsto \kappa_1(*),$$
$$\kappa_2(x', g) \mapsto g(x'),$$
$$\kappa_3(x', e) \mapsto \kappa_3(x', e)\ \}.$$

This J will sometimes be called the Java monad.

It is not hard to check that the three monad equations $\mu_A \circ \eta_{J(A)} = \mathrm{id}$, $\mu_A \circ J(\eta_A) = \mathrm{id}$ and $\mu_A \circ J(\mu_A) = \mu_A \circ \mu_{J(A)}$ are satisfied. Notice that the Java monad J incorporates ingredients from three basic computational monads introduced in [38]: the partiality monad $A \mapsto 1 + A$, the exception monad $A \mapsto A + E$ and the side-effect monad $A \mapsto (S \times A)^S$. But J is not obtained via composition from these basic monads, but from the associated monad transformers [37,8,35], namely $T \mapsto 1 + T(-)$, $T \mapsto T((-) + E)$ and $T \mapsto T(S \times (-))^S$.

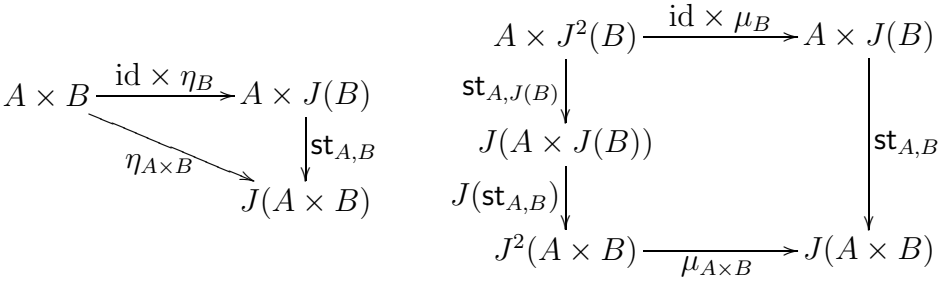
4.1 Extension and composition

It is folklore knowledge that every functor $F\colon$ **Sets** $\to$ **Sets** is strong, with strength natural transformation $\mathsf{st}_{A,B}\colon A \times F(B) \to F(A \times B)$ given by $(a, z) \mapsto$

$F(\lambda b \in B.\,(a,b))(z)$. This strength definition applies in particular to the above functor J. Explicitly,

$$\begin{aligned}\mathsf{st}_{A,B}(a,f) = \lambda x \in S.\ \mathsf{CASES}\ f(x)\ \mathsf{OF}\ \{ \\ \kappa_1(*) \mapsto \kappa_1(*), \\ \kappa_2(x',b) \mapsto \kappa_2(x',(a,b)), \\ \kappa_3(x',e) \mapsto \kappa_3(x',e)\ \}.\end{aligned} \tag{4}$$

In order to show that J is a *strong monad*, and not just a strong functor, we have to check that additionally the following two diagrams commute.



This is an easy exercise. In fact, for a monad on **Sets**, the strength map is uniquely determined, see [38, Proposition 3.4].

Using this strength map there is a standard way to turn functions $f\colon A \times B \to J(C)$ into functions $f^*\colon A \times J(B) \to J(C)$, namely as:

$$f^* = \mu_C \circ J(f) \circ \mathsf{st}_{A,B}. \tag{5}$$

Explicitly, this "Kleisli extension" can be described on $a \in A$ and $g \in J(B)$ as:

$$\begin{aligned}f^*(a,g) = \lambda x \in S.\ \mathsf{CASES}\ g(x)\ \mathsf{OF}\ \{ \\ \kappa_1(*) \mapsto \kappa_1(*), \\ \kappa_2(x',b) \mapsto f(a,b)(x'), \\ \kappa_3(x',e) \mapsto \kappa_3(x',e)\ \}.\end{aligned}$$

Thus, this Kleisli extension is the same as extension for Java described in (2).

For convenience, we shall also describe extension in the other parameter:

$$\begin{aligned}f^\sharp &= (f \circ \mathsf{swap})^* \circ \mathsf{swap} \\ &= \mu_C \circ J(f \circ \mathsf{swap}) \circ \mathsf{st}_{B,A} \circ \mathsf{swap} :\ J(A) \times B \longrightarrow J(C).\end{aligned} \tag{6}$$

where the function swap exchanges the arguments: $\mathsf{swap}(x, y) = (y, x)$. As an aside, a monad is called distributive (see [27]) if evaluation in the first argument and in the second one commute. This is not the case for J. Moggi [38] introduces a special 'let' notation for extension, which is convenient in computations. Here we are merely interested in showing how the monad extension plays a role in Java.

Example 2 *Recall that Java (like C) has two conjunctions, namely "and"* `&` *and "conditional and"* `&&`*, see [11, §§ 15.23]. The first one (*`&`*) always evaluates both arguments, but the second one (*`&&`*) only does so if the first argument evaluates to true. The difference is relevant in the presence of side-effects, non-termination, or exceptions. We show how the two conjunction operations can be described concisely via extension w.r.t. the Java monad.*

First, the one-step extension $(\eta_{\mathsf{bool}} \circ \wedge)^* \colon \mathsf{bool} \times J(\mathsf{bool}) \to J(\mathsf{bool})$ *only has to evaluate its second argument. Extending it again, now in the first argument, yields:*

$$J(\mathsf{bool}) \times J(\mathsf{bool}) \xrightarrow{\&\ \stackrel{def}{=}\ ((\eta_{\mathsf{bool}} \circ \wedge)^*)^\sharp} J(\mathsf{bool}).$$

Explicitly,

$$\begin{aligned} f_1 \,\&\, f_2 = \lambda x \in S.\ \mathit{CASES}\ f_1(x)\ \mathit{OF}\ \{ & \\ & \kappa_1(*) \mapsto \kappa_1(*), \\ & \kappa_2(x_1, b_1) \mapsto\ \mathit{CASES}\ f_2(x_1)\ \mathit{OF}\ \{ \\ & \qquad \kappa_1(*) \mapsto \kappa_1(*), \\ & \qquad \kappa_2(x_2, b_2) \mapsto \kappa_2(x_2, b_1 \wedge b_2), \\ & \qquad \kappa_3(x_2, e_2) \mapsto \kappa_3(x_2, e_2)\,\}, \\ & \kappa_3(x_1, e_1) \mapsto \kappa_3(x_1, e_1)\,\}. \end{aligned}$$

The conditional and `&&` *can be obtained by extending the auxiliary function* $t \colon \mathsf{bool} \times J(\mathsf{bool}) \to J(\mathsf{bool})$ *given by* $t(b, f) = \mathit{IF}\ b\ \mathit{THEN}\ f\ \mathit{ELSE}\ \eta_{\mathsf{bool}}(\mathsf{false})$.

$$J(\mathsf{bool}) \times J(\mathsf{bool}) \xrightarrow{\&\& \stackrel{def}{=} t^\sharp} J(\mathsf{bool}).$$

This concludes the example.

We turn to the Kleisli category $\mathbf{Kl}(J)$ of the Java monad J. Its objects are sets, and its morphisms $A \to B$ are functions $A \to J(B)$. The identity map $A \to J(A)$ in $\mathbf{Kl}(J)$ is the unit η_A at A, and the "Kleisli" composition $g \bullet f \colon A \to J(C)$ of two morphisms $f \colon A \to J(B)$ and $g \colon B \to J(C)$ in $\mathbf{Kl}(J)$ is defined as

usual as:

$$g \bullet f = \mu_C \circ J(g) \circ f. \tag{7}$$

Unraveling yields for $a \in A$,

$$\begin{aligned}(g \bullet f)(a) = \lambda x \in S.\ \mathsf{CASES}\ f(x)\ \mathsf{OF}\ \{ \\ \kappa_1(*) \mapsto \kappa_1(*), \\ \kappa_2(x', b) \mapsto g(b)(x'), \\ \kappa_3(x', e) \mapsto \kappa_3(x', e)\ \}.\end{aligned}$$

Thus the Kleisli composition $\bullet$ is basically the same as Java composition ; from (1): if f does not terminate or terminates abruptly, so does $g \bullet f$, and if f terminates normally and produces a successor state, then g is executed on this state. We thus use a compositional (denotational) semantics: for Java statements s_1, s_2,

$$[\![\, s_1\,;\, s_2 \,]\!] = [\![\, s_2 \,]\!] \bullet [\![\, s_1 \,]\!].$$

The other Java language constructs are interpreted in essentially the same way, see [12].

For future use we define how to iterate a function $s\colon A \to J(A)$ using Kleisli composition:

$$s^n = \begin{cases} \eta_A & \text{if } n = 0 \\ s \bullet s^{n-1} & \text{otherwise.} \end{cases} \tag{8}$$

4.2 Cppo structure on Kleisli homsets

The homsets of the Kleisli category of the Java monad J are the sets of morphisms $f\colon A \to B$ in the Kleisli category $\mathbf{Kl}(J)$, *i.e.* the sets of functions $f\colon A \to J(B)$. Each set $J(B)$ can be ordered via the "pointwise flat" ordering, obtained from the flat ordering on $1+(S\times A)+(S\times E)$ by taking the pointwise extension to $J(B) = (1 + (S \times B) + (S \times E))^S$, which can in turn be extended – pointwise – to functions $A \to J(B)$. Explicitly, for $f_1, f_2\colon A \to J(B)$, $f_1 \sqsubseteq f_2$ if for each $a \in A$ and $x \in S$, $f_1(a)(x)$ hangs, or else is equal to $f_2(a)(x)$. More formally:

$$f_1 \sqsubseteq f_2 \iff \forall a \in A.\, \forall x \in S.\, f_1(a)(x) = \kappa_1(*) \vee f_1(a)(x) = f_2(a)(x). \tag{9}$$

It is not hard to see that $\sqsubseteq$ is a partial ordering. Also, there is a least element $\bot = \lambda a \in A.\, \lambda x \in S.\, \kappa_1(*)$, namely the statement that always hangs. Notice that $f \bullet \bot = \bot$, but $\bot \bullet f$ may be different from $\bot$, namely when f throws an exception.

It is standard that the flat ordering is a complete partial ordering (cpo): an ordering in which each ascending sequence has a least upperbound. Hence the pointwise flat ordering $\sqsubseteq$ also makes the set of morphisms $A \to J(B)$ in the Kleisli category $\mathbf{Kl}(J)$ a cpo. Explicitly, for an ascending chain $(f_n \colon A \to J(B))_{n\in\mathbb{N}}$ there is a least upperbound $f = \bigsqcup_{n\in\mathbb{N}} f_n \colon A \to J(B)$ given by:

$$f(a)(x) = \begin{cases} \kappa_1(*) & \text{if } \forall n \in \mathbb{N}.\, f_n(a)(x) = \kappa_1(*) \\ f_\ell(a)(x) & \text{else, where } \ell \text{ is the least } n \text{ with } f_n(a)(x) \neq \kappa_1(*). \end{cases}$$

The cpo structure together with the bottom element $\bot$ makes each homset a complete pointed partial ordering (cppo). In the category of cppos, morphisms are continuous functions that preserve the ordering, but which need not preserve the bottom element. This is exactly what Kleisli composition $\bullet$ does. Therefore the Kleisli category $\mathbf{Kl}(J)$ is cppo-enriched, see [9].

We summarise what we have found so far.

Proposition 3 *The functor J from (3) describing the outputs of Java statements and expressions is a strong monad on the category of sets. Its Kleisli composition and extension correspond to composition and extension in Java. And its Kleisli category $\boldsymbol{Kl}(J)$ is cppo-enriched.* □

The following result about extension and continuity will be useful later in Subsection 6.2.

Lemma 4 *Consider a function $f \colon A \times B \to J(C)$ and its extension $f^* \colon A \times J(B) \to J(C)$ from (5).*

(1) For each $a \in A$, the function $f^*(a, -) \colon J(B) \to J(C)$ is continuous.
(2) If the set A carries an ordering in such a way that for each $b \in B$, the function $f(-, b) \colon A \to J(C)$ is continuous, then for each $g \in J(B)$, $f^*(-, g) \colon A \to J(C)$ is also continuous. □

5 Other examples

The aim of this section is to briefly illustrate that the situation of Proposition 3 is not uncommon in semantics. The following two functors $F \colon \mathbf{Sets} \times \mathbf{Sets} \to$

Sets are also such that the associated functors T given by $T(B) = F(S,B)^S$ are strong monads, whose Kleisli category $\mathbf{Kl}(T)$ is cppo-enriched.

(1) $F(S,B) = 1+(S\times B)$, describing possibly non-terminating computations. The ordering on the associated Kleisli homsets is again the "pointwise flat" ordering, obtained from the flat ordering on $F(S,B)$ by taking the pointwise extension on $T(B) = F(S,B)^S$.
(2) $F(S,B) = \mathcal{P}(S \times B)$, describing non-deterministic computations (or B-labeled transition systems). The ordering in this case is the pointwise inclusion ordering on subsets.

Also, the analogues of Lemma 4 hold for these T (instead of J).

6 While statements and recursive statements

In this section we assume an arbitrary strong monad T on the category of sets, whose Kleisli category $\mathbf{Kl}(T)$ is cppo-enriched and satisfies the analogue of Lemma 4 (with T instead of J). For such a T we shall give the standard denotational definitions for while statements and for recursion. For the special case where T is the Java monad J we show that these denotational definitions coincide with (a simplified version of) more operational characterisations that are used within the LOOP project. In the simplification only exceptions can cause a break out of a repetition. In Java one may have a `continue` or `break` statement inside a while loop, causing a skip of the remainder of the current cycle (for `continue`), or of the whole while loop altogether (for `break`). The full version of the while statement that is used for the verification of Java programs is described in [14].

6.1 While statement

So let T be our strong monad, with statements interpreted as maps $A \to T(B)$, describing T-computations. The composition of such statements is described by the Kleisli composition $\bullet$ associated with T. For a Boolean computation $c \in T(\mathsf{bool})$ we shall define a function $\mathsf{while}(c)$ taking a statement $s\colon A \to T(A)$ in the Kleisli category $\mathbf{Kl}(T)$ to a new statement $\mathsf{while}(c)(s)\colon A \to T(A)$. This requires a conditional statement, which is defined as follows.

Definition 5 *The conditional operator*

$$T(\mathsf{bool}) \times (T(B)^A)^2 \xrightarrow{\mathit{IfThenElse}} T(B)^A$$

is obtained by extension as $\mathsf{ifthenelse}^\sharp$ *of the obvious map*

$$\mathsf{bool} \times (T(B)^A)^2 \xrightarrow{\quad \mathsf{ifthenelse} \quad} T(B)^A.$$

Thus, IfThenElse *evaluates the condition in* $T(\mathsf{bool})$ *and decides on the basis of its outcome whether to evaluate the first or second statement in the product* $(T(B)^A)^2$*. By Lemma 4 (2)—actually, by its analogue for* T*, plus Currying—the conditional statement* IfThenElse *is continuous in each of its arguments.*

For the case where T is the Java monad J, the conditional statement may be described explicitly as:

$$\begin{aligned}\mathsf{IfThenElse}(c, s_1, s_2) = \lambda a \in A.\, \lambda x \in S.\ \mathsf{CASES}\ c(x)\ \mathsf{OF}\ \{ & \\ \kappa_1(*) &\mapsto \kappa_1(*), \\ \kappa_2(x', b) &\mapsto \mathsf{IF}\ b \\ &\qquad \mathsf{THEN}\ s_1(a)(x') \\ &\qquad \mathsf{ELSE}\ s_2(a)(x'), \\ \kappa_3(x', e) &\mapsto \kappa_3(x', e)\ \}.\end{aligned}$$

In the standard elementary semantics of programming languages using partial functions as denotations of statements, the while statement is characterised as a least fixed point, see *e.g.* [48, Definition 9.18]. This approach may be generalised to our current setting with statements w.r.t. the monad T.

Definition 6 *For a condition* $c \in T(\mathsf{bool})$ *and a statement* $s\colon A \to T(A)$*, a while statement* $\mathsf{while}(c)(s)\colon A \to T(A)$ *is defined as the least fixed point of the operator* $H(c, s)\colon T(A)^A \to T(A)^A$ *given by:*

$$H(c, s) = \lambda t \in T(A)^A.\, \mathsf{IfThenElse}(c, t \bullet s, \eta_A).$$

This operator $H(c, s)$ *is continuous, because both* $\bullet$ *and* IfThenElse *are continuous.*

We proceed with the operational description of while statements $\mathsf{while}(c)(s)$ for the Java monad J. The idea in this case is to iterate the statement s until the condition c becomes false. But there are various subtleties involved:

(1) The condition c may itself have a side-effect, which has to be taken into account. Therefore we iterate $s \bullet \widehat{c}$, where $\widehat{c}\colon A \to J(A)$ is the statement

which executes c only for its side-effect and ignores its result:

$$\begin{aligned}\widehat{c}(a)(x) = \ & \mathsf{CASES}\ c(x)\ \mathsf{OF}\ \{ \\ & \kappa_1(*) \mapsto \kappa_1(*), \\ & \kappa_2(x', b) \mapsto \kappa_2(x', a), \\ & \kappa_3(x', e) \mapsto \kappa_3(x', e)\ \}.\end{aligned}$$

Or, equivalently, $\widehat{c}(a) = J(\lambda b \in \mathsf{bool}.\, a)(c)$.

(2) During the iteration both c and s may throw exceptions. If this happens the while statement must throw the same exception.

In order to detect that the condition becomes false or an exception is thrown two partial functions $N(c,s), E(c,s) : A \to \mathbb{N}^S$ are defined. The number $N(c,s)(a)(x)$, if defined, is the smallest number of iterations after which c becomes false without occurrence of exceptions. Similarly, $E(c,s)(a)(x)$, if defined, is the smallest number of iterations after which an exception is thrown. More formally, $N(c,s)(a)(x)$ is the smallest number n such that

$$(\widehat{c} \bullet (s \bullet \widehat{c})^n)(a)(x) = \kappa_2(x', \mathsf{false})$$

for some x', if such a number n exists, and $E(c,s)(a)(x)$ is the smallest number n such that

$$(s \bullet \widehat{c})^n(a)(x) = \kappa_3(x', e)$$

for some x' and e, if such a number n exists.

By the definition of Kleisli composition, if $f(a)(x) = \kappa_3(x', e)$ for some $f : A \to J(B)$ then $(g \bullet f)(a)(x) = f(a)(x)$ for all $g : B \to J(C)$. In other words, if $f(a)(x)$ throws an exception, then $(g \bullet f)(a)(x)$ also throws that exception. So, if $(s \bullet \widehat{c})^n(a)(x)$ throws an exception then $\widehat{c} \bullet (s \bullet \widehat{c})^m(a)(x)$ throws the same exception for all $m \geq n$. This means that if both $N(c,s)(a)(x)$ and $E(c,s)(a)(x)$ are defined, then $N(c,s)(a)(x)$ is smaller than $E(c,s)(a)(x)$. This is used in the following operational description of the while statement for the Java monad.

Proposition 7 *For a condition $c \in J(\mathsf{bool})$ and a statement $s\colon A \to J(A)$ the while statement $\mathsf{while}(c)(s)\colon A \to J(A)$ from Definition 6 can equivalently be described as:*

$$\mathsf{while}(c)(s) = \lambda a \in A.\, \lambda x \in S.$$

$$\begin{cases} (\widehat{c} \bullet (s \bullet \widehat{c})^n)(a)(x) & \text{if } N(c,s)(a)(x) = n \\ (s \bullet \widehat{c})^n(a)(x) & \text{if } E(c,s)(a)(x) = n \text{ and } N(c,s)(a)(x) \text{ is undefined} \\ \kappa_1(*) & \text{if both } N(c,s)(a)(x) \text{ and } E(c,s)(a)(x) \text{ are undefined.} \end{cases}$$

PROOF. The proof that $\mathsf{while}(c)(s)$ is a fixed point of the operator $H(c,s)$ from Definition 6 proceeds by distinguishing many cases and handling them one by one.

First we consider the following three cases: (1) the condition hangs; (2) the condition throws an exception; (3) the condition terminates normally and evaluates to false. In all these cases the statement is not executed at all, and the outcome of the while is easily established using the above functions N and E: it hangs in case of (1), it throws the same exception as the condition in (2) and it terminates normally in (3).

The statement does get executed in case: (4) the condition terminates normally and evaluates to true. This leads to the subcases: (4a) the statement hangs; (4b) the statement throws an exception; (4c) the statement terminates normally. In the last case we use the following auxiliary result: if $s(a)(x') = \kappa_2(x'', a')$ and $c(x) = \kappa_2(x', \mathsf{true})$, then $\mathsf{while}(c)(s)(a)(x) = \mathsf{while}(c)(s)(a')(x'')$.

In order to show that $\mathsf{while}(c)(s)$ is the *least* fixed point of $H(c,s)$, we assume a function $t\colon A \to J(A)$ with $H(c,s)(t) = t$. We then first show by induction on n that:

(1) If $N(c,s)(a)(x) = n$, then $t(a)(x) = (\widehat{c} \bullet (s \bullet \widehat{c})^n)(a)(x)$.
(2) If $E(c,s)(a)(x) = n$, then $t(a)(x) = (s \bullet \widehat{c})^n(a)(x)$.

The result $\mathsf{while}(c)(s) \sqsubseteq t$ then follows by unpacking the definition of while. □

Definition 6 and Proposition 7 give quite different characterisations of the meaning of while. Both correspond to an intuitive understanding of the semantics of while. Which of these characterisations is the more fundamental one – and should therefore be considered as the definition – is a matter of taste, but we should prove their equivalence. In reasoning about while repetitions we often found the definition given by Proposition 7 more convenient than the more conventional Definition 6.

A similar analysis can be given for Java's `for` statement. What is slightly more difficult is that an extra local variable with update function has to be taken into account.

6.2 Recursive statements

Again we start from the general situation with a computational monad T. Recursive statements with input type A and result type B are interpreted as functions of type $A \to T(B)$, which are constructed from a (continuous) map-

ping from statements to statements of type $T(B)^A \to T(B)^A$. The semantics of such a recursive statement can be defined in the same way as the semantics of the while statement.

Definition 8 *For a continuous mapping $H\colon T(B)^A \to T(B)^A$ we define the statement $\mathsf{rec}(H)\colon A \to T(B)$ as the least fixed point of H.*

For the special case where T is the Java monad J, we can use the explicit description of the least fixed point from Subsection 4.2 to get the expected operational description of recursive statements (as used in the LOOP project): for a mapping H as above a partial function $U(H) : A \to \mathbb{N}^S$ is defined. The number $U(H)(a)(x)$, if it exists, is the smallest n such that $H^n(\bot)(a)(x) \neq \kappa_1(*)$. Then

$$\mathsf{rec}(H) = \lambda a \in A.\, \lambda x \in S.\, \begin{cases} H^n(\bot)(a)(x) & \text{if } U(H)(a)(x) = n \\ \kappa_1(*) & \text{if } U(H)(a)(x) \text{ is undefined.} \end{cases}$$

Continuity of a particular H is typically easy to show. By Lemma 4 all extensions—like `&` and `&&` in Example 2—are continuous, and it is a standard result that application, lambda abstraction, composition, and the taking of least fixed points all preserve continuity.

7 Logic for coalgebras

The new research area of modal and temporal logic for coalgebras is fairly active [39,46,18,19,21,2,47,29]. Most of these studies work with coalgebras $X \xrightarrow{c} T(X)$ of so-called polynomial functors T. These are built up inductively, using identity and constant functors, products and coproducts, exponents (with a constant set), and possibly also powersets. All the functors we have considered so-far are polynomial. The results obtained in this area show that temporal and modal logics are the natural logics for coalgebras, by providing tailor made next-time operators for reasoning about the dynamical behaviour of coalgebras. These results extend standard results from modal logic, like completeness and Hennessey-Milner style characterisation of bisimilarity, namely as validity for all formulas of the logic. Here we shall not go into this theory in general, but sketch some of its consequences for the semantics of Java. The temporal and modal operators that we describe are as in [18,19].

The dual, monadic/coalgebraic picture that we have described for the semantics of Java has concentrated on the computational structure obtained from the monad $J(A) = F(S, A)^S$, for $F(S, A) = 1+(S\times A)+(S\times E)$. From now on we shall take a coalgebraic look and consider the functor $L(S) = F(S, B)^A$,

for fixed sets A, B. Java statements are then coalgebras $S \to L(S)$ of this functor L. This coalgebraic view allows variation in the state-space S, leading to notions such as coalgebra homomorphism, invariance, bisimilarity, modal operators, and Hoare logic. This will be sketched below.

7.1 Modal operators for Java

Modal operators can be defined for coalgebras of polynomial functors by induction on the structure of the functor, see [18,19]. Therefore one distinguishes in the result type of a coalgebra c occurrences of the state space S, and of constants A, as in:

$$S \xrightarrow{\quad c \quad} \boxed{\cdots S \cdots A \cdots S \cdots}$$

For each occurrence of S in the result type (box) one has an operator that acts on predicates on S. It maps a predicate $P \subseteq S$ to a new predicate consisting of those states $x \in S$ such that if $c(x)$ yields a result state, say y, at the occurrence of S we are talking about, then $P(y)$ holds. This operator thus says "next-time P at this occurrence". Similarly, for each occurrence of a constant A in the result type (not as exponent) there is an "observer" yielding atomic predicates. It maps an element $a \in A$ to the predicate consisting of those $x \in S$ such that if $c(x)$ yields a result observation at the occurrence A that we are talking about, then this observation is a.

In order to make this more concrete, we shall consider these operators and observers for the functor L, starting from a given statement s, as in:

$$S \xrightarrow{\quad s \quad} \Big(1 + (S \times B) + (S \times E)\Big)^A.$$

The two occurrences of S on the right-hand side give rise to two next-time operators: Ns, for normal state, and Es, for exceptional state. They are defined on a predicate $P \subseteq S$ and an input $a \in A$ as:

$$\mathsf{Ns}(P)(a) = \{x \in S \mid \forall x', b'.\, s(x)(a) = \kappa_2(x', b') \Rightarrow P(x')\}$$

$$\mathsf{Es}(P)(a) = \{x \in S \mid \forall x', e'.\, s(x)(a) = \kappa_3(x', e') \Rightarrow P(x')\}.$$

Thus, $\mathsf{Ns}(P)(a)$ holds for those states for which, if the statement with input a terminates normally, then P holds in the result state. Note that these operators Ns and Es implicitly depend on the statement s. We could make this explicit by writing $\mathsf{Ns}[s](P)(a)$ and $\mathsf{Es}[s](P)(a)$ instead—like in dynamic logic.

Similarly, the three (non-exponent) constants 1, B, E give rise to three observer predicates, Hg, $\mathsf{Rs}(b)$, $\mathsf{Ex}(e)$, (for hang, result, and exception, resp.)

defined for $a \in A$ as:

$$\begin{aligned}
\mathsf{Hg}(a) &= \{x \in S \mid s(x)(a) = \kappa_1(*)\} \\
\mathsf{Rs}(b)(a) &= \{x \in S \mid \forall x', b'.\, s(x)(a) = \kappa_2(x', b') \Rightarrow b' = b\} \\
\mathsf{Ex}(e)(a) &= \{x \in S \mid \forall x', e'.\, s(x)(a) = \kappa_3(x', e') \Rightarrow e' = e\}.
\end{aligned}$$

Like the operators Ns and Es earlier the observers Hg, $\mathsf{Rs}(b)$, $\mathsf{Ex}(e)$ implicitly depend on the statement s; this could be make explicit in the same way.

With these operators and observers one can express various properties. For example, let false be the constant predicate "false"; the predicate $\mathsf{Ns}(\mathsf{false})(a) \wedge \mathsf{Es}(\mathsf{false})(a)$ then describes those states where our statement s hangs at a. And $\neg\mathsf{Ns}(\mathsf{false})(a)$ contains the states where s *must* terminate normally (with input a). Also, these operators interact appropriately with the Kleisli composition structure of the monad J. For instance, there is a "normal" composition rule for Ns, of the form

$$\mathsf{Ns}[s_1\,;s_2](P)(a) = \bigvee_{b \in B} \mathsf{Rs}[s_1](b)(a) \;\wedge\; \mathsf{Ns}[s_1](\mathsf{Ns}[s_2](P)(b))(a). \tag{10}$$

This provides a suitable connection between the computational structure provided by the monad, and the logical structure provided by the functor of the coalgebra. How to obtain such connections for more general Kleisli maps / coalgebras is still an open question.

A predicate $P \subseteq S$ is called an *invariant* (see also [17,18]) if for all $a \in A$,

$$P \Longrightarrow \mathsf{Ns}(P)(a) \wedge \mathsf{Rs}(P)(a).$$

This means that s preserves P: once P holds, it will continue to hold no matter how many times one applies s.

Writing $\Box P$ for the greatest invariant in P, we can read $\Box P$ as "always P" or "henceforth P". The $\Box$ operator is useful for expressing safety properties—and its dual $\Diamond = \neg\Box\neg$ for liveness properties.

What we see is that appropriate logical operators for reasoning about Java statements can be obtained via a structural analysis of the functor that captures these statements as coalgebras.

In this final part we illustrate how the logical operators from the previous section given rise to a suitable Hoare logic for Java. The "obvious" way to do this is to introduce separate Hoare triples for the different termination modes, as in [14,12]. For predicates $P, Q \subseteq S$ write for example:

$$\{P\}\, s(a)\, \{\mathsf{normal}(Q, b)\} \qquad \text{for} \qquad P \Longrightarrow \mathsf{Ns}(Q)(a) \wedge \mathsf{Rs}(b)(a)$$

$$\{P\}\, s(a)\, \{\mathsf{exceptional}(Q, e)\} \qquad \text{for} \qquad P \Longrightarrow \mathsf{Es}(Q)(a) \wedge \mathsf{Ex}(e)(a).$$

These are partial correctness triples. The first one says that if the precondition P holds and if $s(a)$ terminates normally, then the postcondition Q holds and the result value is equal to b.

Associated with these definitions there are appropriate rules, like:

$$\frac{\{P\}\, s_1(a)\, \{\mathsf{normal}(Q, b)\} \qquad \{Q\}\, s_2(b)\, \{\mathsf{exceptional}(R, e)\}}{\{P\}\, (s_1\, ;\, s_2)(a)\, \{\mathsf{exceptional}(R, e)\}} \tag{11}$$

Such rules illustrate an interaction like in (10) above between the computational structure given by the monad, and the logical structure given by the functor of the coalgebras.

Things get more interesting if we consider the annotation language JML [32] for Java. This language involves correctness assertions which can be added as special comments to Java classes. For example, the behaviour of a Java method `m` may be described in JML as:

```
/*@ behavior
  @     diverges: D  <pre-condition for non-termination>
  @     requires: P  <precondition>
  @   modifiable: M  <fields that can be modified>
  @      ensures: Q  <postcondition for normal termination>
  @ signals(E e): R  <postcondition for exceptional termination>
  @*/
B m(A a) { ... }
```

The meaning of such behaviour specifications can be expressed via the logical operators and observers described above. This is slightly complicated by the fact that the `Q` and `R` above are predicates involving not only the post-state, but also the pre-state—for evaluation of expressions like `\old`(e) referring to the value of `e` in the state before the method is evaluated. But this can be handled in the standard way by using what are called logical variables (the y, b, e below). Ignoring modifies clauses (and class invariants, which should be

added to the pre- and post-conditions), the meaning of the above specification can be expressed as:

$$\begin{array}{l}\forall a \in [\![\, A \,]\!].\, \forall b \in [\![\, B \,]\!].\, \forall e \in [\![\, E \,]\!].\, \forall y \in S.\\ \quad [\![\, \mathtt{P} \,]\!] \wedge (y = \_)\\ \qquad \Longrightarrow\\ \quad \Big(\mathsf{Hg}(a) \Rightarrow [\![\, \mathtt{D} \,]\!]\Big) \wedge \Big(\mathsf{Rs}(b)(a) \Rightarrow \mathsf{Ns}([\![\, \mathtt{Q} \,]\!](y,b))(a)\Big) \wedge\\ \qquad \Big(\mathsf{Ex}(e)(a) \Rightarrow \mathsf{Es}([\![\, \mathtt{R} \,]\!](y,e))(a)\Big)\end{array}$$

where $(y = \_)$ is the predicate $\{x \mid x = y\}$. This formula says that if in a state equal to y the precondition holds, then (1) if the statement hangs with input a, then (the translation of) D holds (in the original pre-state), (2) if the statement with input a terminates normally, then in the post-state $\mathtt{Q}(y, b)$ holds, where y is the pre-state and b the result value, and (3) if the statement with input a terminates with an exception e, then $\mathtt{Q}(y, e)$ holds in the post-state.

For more information on the logical semantics of JML we refer to [23], where appropriate proof rules are described for such JML specifications, following the computational structure (like in (10) and (11) above). These rules also involve the extra abnormalities (for return, break and continue, as described in Section 3) because they may arise during computations—but they cannot emerge from a method, and so they are not mentioned in JML.

8 Conclusions

We have investigated the structure at the heart of the denotational and axiomatic semantics of sequential Java developed in the LOOP project, which is used as the basis of mechanically assisted verification of actual Java programs. We have described this structure both from a monadic and from a coalgebraic point of view. The monadic point of view, investigated in Sections 4 to 6, provides a monad that is useful for organising the computational structure of the Java semantics, particularly when it comes to all the subtleties involved with handling abnormal termination. The coalgebraic point of view, investigated in Section 7, provides suitable modal operations for reasoning about this computational structure, and with which in particular a Hoare logic for Java can be defined.

The monadic/coalgebraic approach is quite general, in that the same basic machinery is used to deal with non-termination and abrupt termination. As mentioned in Section 5, other computational features, such as non-determinism, could be dealt with in a similar way.

It is interesting to contrast our approach to the one taken in the denotational semantics of sequential Java described in [1]. In our approach the monad J makes explicit everything involved with the control flow of programs at the level of types. In the approach of [1] some of this complexity is implicit in the shape of the global state S, which contains global variables that keep track of exceptions and breaks. A disadvantage of this approach is that in the definition of the semantics one has to be very careful to update this information in all the right places; forgetting to do this at some point will mean that a wrong continuation will be taken. Given the size and complexity of the semantics, avoiding such mistakes is not trivial [6]. In our approach the type information of the very basic type system discussed in Section 2 provides some safety: for the definitions to be well-typed one is essentially forced to consider all options, and simply forgetting a case would result in an ill-typed rather than an incorrect definition. This advantage is in particular relevant for the LOOP compiler, as it provides denotations as PVS or Isabelle code, which can indeed be mechanically typechecked.

Acknowledgements

Thanks are due to the anonymous referees whose sharp comments greatly improved the quality of the paper.

References

[1] J. Alves-Foss and F. S. Lam. Dynamic denotational semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 in Lecture Notes in Computer Science, pages 201–240. Springer, Berlin, 1998.

[2] A. Baltag. A logic for coalgebraic simulation. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, volume 33 in Electronic Notes in Theoretical Computer Science, Elsevier, Amsterdam, 2000.

[3] M. Barr and Ch. Wells. *Toposes, Triples and Theories*. Springer, Berlin, 1985.

[4] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, volume 1827 in Lecture Notes in Computer Science, pages 1–21. Springer, Berlin, 2000.

[5] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and*

[6] For example, the definition of the semantics of the while statement in [1] does not appear to update the "continue" information when a repetition is entered.

Analysis of Software (TACAS), volume 2031 in Lecture Notes in Computer Science, pages 299–312. Springer, Berlin, 2001.

[6] J. van den Berg, B. Jacobs, and E. Poll. Formal specification and verification of JavaCard's Application Identifier Class. In I. Attali and Th. Jensen, editors, *Java on Smart Cards: Programming and Security*, volume 2041 in Lecture Notes in Computer Science, pages 137–150. Springer, Berlin, 2001.

[7] F. Christian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10(2):163–174, 1984.

[8] D. A. Espinosa. *Semantic Lego.* PhD thesis, Colombia Univ., 1995.

[9] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* Cambridge Univ. Press, 1996.

[10] R. Goldblatt. *Logics of Time and Computation.* CSLI Lecture Notes 7, Stanford, 2^{nd} rev. edition, 1992.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, 2000.

[12] M. Huisman. *Reasoning about JAVA Programs in higher order logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.

[13] M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1869 in Lecture Notes in Computer Science, pages 301–319. Springer, Berlin, 2000.

[14] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 in Lecture Notes in Computer Science, pages 284–303. Springer, Berlin, 2000.

[15] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. *Int. Journ. on Software Tools for Technology Transfer*, 3(3):332–352, 2001.

[16] B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.

[17] B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, volume 1349 in Lecture Notes in Computer Science, pages 276–291. Springer, Berlin, 1997.

[18] B. Jacobs. The temporal logic of coalgebras via Galois algebras. Techn. Rep. CSI-R9906, Comput. Sci. Inst., Univ. of Nijmegen. To appear in *Mathematical Structures in Computer Science*, 1999.

[19] B. Jacobs. Towards a duality result in coalgebraic modal logic. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, volume 33 in Electronic Notes in Theoretical Computer Science, Elsevier, Amsterdam, 2000.

[20] B. Jacobs. A formalisation of Java's exception mechanism. In D. Sands, editor, *Programming Languages and Systems (ESOP)*, volume 2028 in Lecture Notes in Computer Science, pages 284–301. Springer, Berlin, 2001.

[21] B. Jacobs. Many-sorted coalgebraic modal logic: a model-theoretic study. *Inf. Théor. et Appl.*, 35(1):31–59, 2001.

[22] B. Jacobs and E. Poll. A monad for basic Java semantics. In T. Rus, editor, *Algebraic Methodology and Software Technology*, volume 1816 in Lecture Notes in Computer Science, pages 150–164. Springer, Berlin, 2000.

[23] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 in Lecture Notes in Computer Science, pages 284–299. Springer, Berlin, 2001.

[24] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[25] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.

[26] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Principles of Programming Languages*, pages 71–84. ACM Press, 1993.

[27] A. Kock. Strong functors and monoidal monads. *Arch. Math.*, XXIII:113–120, 1972.

[28] S. Krstić, J. Launchbury, and D. Pavlović. Categories of processes enriched in final coalgebras. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures*, volume 2030 in Lecture Notes in Computer Science, pages 303–317. Springer, Berlin, 2001.

[29] A. Kurz. Specifying coalgebras with modal logic. *Theor. Comp. Sci.*, 260(1-2):119–138, 2001.

[30] S. Mac Lane. *Categories for the Working Mathematician*. Springer, Berlin, 1971.

[31] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.

[32] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ.
`http://www.cs.iastate.edu/~leavens/JML.html`, 1999.

[33] K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995.

[34] K.R.M. Leino and J.L.A. van de Snepscheut. Semantics of exceptions. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 447–466. North-Holland, 1994.

[35] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP'96: 6th European Symposium on Programming*, volume 1058 in Lecture Notes in Computer Science, pages 219–234. Springer, Berlin, 1996.

[36] E.G. Manes. *Algebraic Theories.* Springer, Berlin, 1974.

[37] E. Moggi. An abstract view of programming languages. Techn. rep. LFCS-90-113, Univ. Edinburgh, 1990.

[38] E. Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.

[39] L.S. Moss. Coalgebraic logic. *Annals of Pure & Applied Logic*, 96(1-3):277–317, 1999. *Erratum* in *Annals of Pure & Applied Logic*, 99(1-3):241–259, 1999.

[40] S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[41] L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and computer science*, pages 361–386. Academic Press, London, 1990. The APIC series, vol. 31.

[42] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application*, pages 135–154. Kluwer Acad. Publ., 2000.

[43] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[44] Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.

[45] H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.

[46] M. Rößiger. Coalgebras and modal logic. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, Volume 33 in Electronic Notes in Theoretical Computer Science, Elsevier, Amsterdam, 2000.

[47] M. Rößiger. From modal logic to terminal coalgebras. *Theoretical Computer Science*, 260(1-2):209–228, 2001.

[48] J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Semantics.* MIT Press, Cambridge, MA, 1977.

[49] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 9:461–493, 1992.