A Monad for Basic Java Semantics *

Bart Jacobs, Erik Poll

Dept. of Computer Science, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{bart, erikpoll}@cs.kun.nl
http://www.cs.kun.nl/~{bart, erikpoll}

Abstract This paper describes the role of a computational monad in the denotational semantics of sequential Java and investigates some of its properties. This denotational semantics is an abstraction of the one used for the verification of (sequential) Java programs using proof tools, see [11,15].

1 Introduction

This paper describes the role of a computational monad in the denotational semantics of sequential Java, that has been developed as part of the LOOP project (Logic of Object-Oriented Programming) [11,15]. This Java semantics provides the basis of 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. The theorem provers currently supported are PVS [13] and Isabelle [14], 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 all of sequential Java, including details such as exceptions, breaks, and nontermination. It has been used to reason about existing Java programs, for instance to prove an invariant property for the Vector class in the Java standard library [8].

We will not describe the denotational semantics of all of the Java constructs in this paper, but concentrate on the use of a monad to 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 nontermination (using the familiar $\perp$), but it also involves different forms of abrupt termination of programs, such as exceptions and the different ways of "jumping" out of methods and repetitions via `break`, `return`, and `continue` statements.

We show that the computational monad approach [12] provides a useful level of abstraction and a good means for organizing all the complications that come

* To appear in: Algebraic Methodology and Software Technology (AMAST), Springer LNCS, 2000.

with defining the semantics of a real programming language like 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.

Interestingly, the Java semantics used in the LOOP project has originally been developed from a coalgebraic perspective [16,9,10]. This perspective focuses on the state space as a black box, and leads to useful notions like class invariant and bisimilarity. The computational monad view is different. It keeps the state space fixed and focuses on the (functional) input-output behaviour. It took us some time to fully appreciate these differences, and to get the most out of both approaches in program semantics.

Not all aspects of the Java semantics are described in this paper; the representation of the global state space, essentially a mapping from references to values, is discussed [2], and the treatment of inheritance is discussed in [6].

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. Its Kleisli composition corresponds to Java composition, and its extension to Java extension. Furthermore, the homsets in its Kleisli category have a cpo structure. Next, in Section 5, while statements and recursive statements are studied in this framework. It is shown how an operational definition of while, taking abnormalities into account, can be described as a least fixed point—like in standard denotational semantics. The cpo structure of Kleisli homsets allows us to deal with recursive statements in the usual way. This whole computational monad approach has (also) been formalised in PVS. This is briefly discussed in the final section 6.

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 $\pi_i : X_1 \times \cdots \times X_n \to X_i$. 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) $\kappa_i : X_i \to X_1 + \cdots + X_n$. There is an associated "case" construction which is perhaps not so familiar: given n functions $f_i : X_i \to Y$, there is a unique function[1] $f : X_1 + \cdots + X_n \to Y$ with $f \circ \kappa_i = f_i$, for all $1 \leq i \leq n$. We shall write

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

This function matches if $z \in X_1 + \cdots + X_n$ is of the form $\kappa_i(x_i)$, and applies in that case f_i to x_i.

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

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.

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

The state space S is a global store giving the values of all program variables. We will not go in to the precise form of the store here; for more detail, see [2]. 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 nontermination (or "hanging").

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

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

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 $\mathsf{StatResult}(S)$ and $\mathsf{ExprResult}(S, B)$, where:

$$\begin{aligned} \mathsf{StatResult}(S) &= 1 + S + \mathsf{StatAbn}(S) \\ \mathsf{ExprResult}(S, B) &= 1 + \big(S \times B\big) + \mathsf{ExprAbn}(S) \end{aligned}$$

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:

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

where $\mathsf{RefType}$ 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 translated 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 translated 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 $S \mapsto \mathsf{ExprResult}(S, B)^{A1 \times \cdots \times An}$.

On the basis of this representation of statements and expressions all language constructs of (sequential) Java are translated into PVS and Isabelle [11,7,6,2,15]. 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:

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

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.

The Java evaluation strategy prescribes that arguments should be evaluated first, from left to right (see [5, §§ 15.6.4]). 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[2] t^* of t with type $t^*\colon \big(S \times (S \to \mathsf{ExprResult}(S, A))\big) \to \mathsf{StatResult}(S)$, namely as:

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

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

(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 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 with result type 1. Thus, our general state transformer functions are of the form:

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

Within the LOOP semantics they are regarded as coalgebras:

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

But here we shall look at them as morphisms

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

in the Kleisli category of a monad. Via currying these two representations are of course equivalent, but they give different perspectives. In the coalgebraic view the state space S plays a central role, in the monadic view state is just one of the ingredients of the monad, like partiality and exceptions.

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

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

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

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

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 monad J incorporates ingredients from three basic computational monads introduced in [12]: 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, so the modular approach to computational monads from *e.g.* [3] is not relevant here.

4.1 Extension and composition

It is folklore knowledge that every functor $F\colon \mathbf{Sets} \to \mathbf{Sets}$ 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,

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

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

$$
\begin{tikzcd}
A \times B \arrow[r, "\mathrm{id} \times \eta_B"] \arrow[dr, "\eta_{A \times B}"'] & A \times J(B) \arrow[d, "\mathsf{st}_{A,B}"] \\
& J(A \times B)
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times J^2(B) \arrow[r, "\mathrm{id} \times \mu_B"] \arrow[d, "\mathsf{st}_{A,J(B)}"'] & A \times J(B) \arrow[dd, "\mathsf{st}_{A,B}"] \\
J(A \times J(B)) \arrow[d, "J(\mathsf{st}_{A,B})"'] & \\
J^2(A \times B) \arrow[r, "\mu_{A \times B}"'] & J(A \times B)
\end{tikzcd}
$$

This is an easy exercise.

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:

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

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

Example 1. Recall that Java (like C) has two conjunctions, namely "and" `&` and "conditional and" `&&`, see [5, §§ 15.22]. 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 both these operations can be obtained via Kleisli extension, starting from the standard conjunction operations $\wedge\colon \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$.

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. Swapping the arguments appropriately—via the function swap with $\mathsf{swap}(x, y) = (y, x)$—and extending again yields:

$$J(\mathsf{bool}) \times J(\mathsf{bool}) \xrightarrow[\big((\eta_{\mathsf{bool}} \circ \wedge)^* \circ \mathsf{swap}\big)^* \circ \mathsf{swap}]{\&\ \stackrel{\mathrm{def}}{=}} J(\mathsf{bool})$$

Explicitly,

$$\begin{aligned} f_1 \ \&\ f_2 = \lambda x \in S.\ \mathsf{CASES}\ f_1(x)\ \mathsf{OF}\ \{ \\ & \kappa_1(u_1) \mapsto \kappa_1(u_1), \\ & \kappa_2(x_1, b_1) \mapsto \mathsf{CASES}\ f_2(x_1)\ \mathsf{OF}\ \{ \\ & \qquad \kappa_1(u_2) \mapsto \kappa_1(u_2), \\ & \qquad \kappa_2(x_2, b_2) \mapsto \kappa_2(x_2, b_1 \wedge b_2), \\ & \qquad \kappa_3(x_2, e_2) \mapsto \kappa_3(x_2, e_2)\ \}, \\ & \kappa_3(x_1, e_1) \mapsto \kappa_3(x_1, e_1)\ \} \end{aligned}$$

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

$$J(\mathsf{bool}) \times J(\mathsf{bool}) \xrightarrow[t^* \circ \mathsf{swap}]{\&\&\ \stackrel{\mathrm{def}}{=}} J(\mathsf{bool})$$

This concludes the example.

We turn to the Kleisli category $\mathbf{Kl}(J)$ of the 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 standardly defined as:

$$g \bullet f = \mu_C \circ J(g) \circ f. \tag{6}$$

Unraveling yields for $a \in A$,

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

Thus, 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.

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

$$s^n = \begin{cases} \eta_A & \text{if } n = 0 \\ s \bullet s^{n-1} & \text{otherwise.} \end{cases} \tag{7}$$

4.2 Cpo structure on Kleisli homsets

The homsets of the Kleisli category of the monad J are the sets of morphisms $f: A \to B$ in the Kleisli category $\mathbf{Kl}(J)$, *i.e.* the sets of functions $f: A \to J(B)$. These can be ordered via the pointwise flat order: for $f_1, f_2: A \to J(B)$, $f_1 \sqsubseteq f_2$ 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:

$$f_1 \sqsubseteq f_2 \iff (\forall a \in A. \forall x \in S. f_1(a)(x) = \kappa_1(*) \vee f_1(a)(x) = f_2(a)(x)) \tag{8}$$

It is not hard to see that $\sqsubseteq$ is a partial order. Also, there is a least element $\bot = \lambda a \in A. \lambda x \in S. \kappa_1(*)$. 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 order is a complete partial order (cpo): an order in which each ascending sequence has a least upperbound. Hence the pointwise flat order $\sqsubseteq$ also makes the set of morphisms $A \to J(B)$ in the Kleisli category $\mathbf{Kl}(J)$ a cpo. Explicitly, for an ascending chain $(f_n: A \to J(B))_{n \in \mathbb{N}}$ there is a least upperbound $f = \bigsqcup_{n \in \mathbb{N}} f_n: A \to J(B)$ given by:

$$f(a)(x) = \begin{cases} \kappa_1(*) & \text{if } \forall n \in \mathbb{N}. f_n(a)(x) = \kappa_1(*) \\ f_\ell(a)(x) & \text{else, where } \ell \text{ is the least } n \text{ with } f_n(a)(x) \neq \kappa_1(*) \end{cases} \tag{9}$$

Kleisli composition $\bullet$ is continuous (*i.e.* preserves the order and least upperbounds) in both its arguments. This means that the Kleisli category $\mathbf{Kl}(J)$ is cpo-enriched, see [4].

We summarise what we have found so far.

Proposition 1. *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 $\mathbf{Kl}(J)$ is cpo-enriched.* □

The following result about extension and continuity is useful later in Subsection 5.2.

Lemma 1. *Consider a function $f: A \times B \to J(C)$ and its extension $f^*: A \times J(B) \to J(C)$ from (5).*

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

5 While statements and recursive statements

This section starts with the definition of the while construct that is used for the semantics of Java. Actually, here we present a simplified version in which exceptions cause a break out of the while statement. In Java one may have a `continue` statement inside a while loop, causing a skip of only the remainder of the current cycle. The full version that is used for the verification of Java programs is described in [7]. The second part of this section is devoted to handling of recursive statements, via least fixed points.

5.1 While statement

For a Boolean computation $c \in J(\mathsf{bool})$ we define a function $\mathsf{while}(c)$ taking a statement $s\colon A \to J(A)$ in the Kleisli category $\mathbf{Kl}(J)$ to a new statement $\mathsf{while}(c)(s)\colon A \to J(A)$. The idea is of course to iterate s until 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:

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

   (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 we define two partial functions $N(c, s), E(c, s) : A \to S \to \mathbb{N}$. 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 definition of the while statement.

Definition 2. *For a condition $c \in J(\mathsf{bool})$ and a statement $s: A \to J(A)$ we define the statement $\mathsf{while}(c)(s): A \to J(A)$ as:*

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

In the standard elementary semantics of programming languages using partial functions the while statement is characterised as least fixed point *e.g.* [17, Definition 9.18]. The next result shows that this also holds in the present setting (with side-effects in expressions and with exceptions).

Proposition 2. *The above while statement $\mathsf{while}(c)(s)$ is the least fixed point of the operator $F(c,s): (A \to J(A)) \to (A \to J(A))$ given by:*

$$F(c,s) = \lambda t \in A \to J(A).\, \mathsf{IfThenElse}(c, t \bullet s, \eta_A)$$

where the conditional statement is defined as:

$$\begin{aligned} \mathsf{IfThenElse}(c, t_1, t_2) = \lambda a \in A.\, \lambda x \in S.\, \mathsf{CASES}\ c(x)\ \mathsf{OF}\ \{ \\ \kappa_1(u) &\mapsto \kappa_1(u), \\ \kappa_2(x', b) &\mapsto \mathsf{IF}\ b \\ &\qquad \mathsf{THEN}\ t_1(a)(x') \\ &\qquad \mathsf{ELSE}\ t_2(a)(x'), \\ \kappa_3(x', e) &\mapsto \kappa_3(x', e)\ \} \end{aligned}$$

This conditional statement may also be defined via extension, applied to the obvious conditional function $(A \to J(A))^2 \times \mathsf{bool} \to (A \to J(A))$.

Proof. The proof that $\mathsf{while}(c)(s)$ is a fixed point of $F(c,s)$ 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 $c(x) = \kappa_2(x', \mathsf{true})$ and $s(a)(x') = \kappa_2(x'', a')$, 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 $F(c, s)$, we assume a function $t: A \to J(A)$ with $F(c, s)(t) = t$. We then first show by induction on n that:

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

5.2 Recursive statements

A recursive method with input type A and result type B is interpreted as a function of type $A \to J(B)$, which is constructed from a mapping from statements to statements of type $(A \to J(B)) \to (A \to J(B))$. The semantics of such a recursive statement can be defined in the same way as the semantics of the while statement. For a mapping $F: (A \to J(B)) \to (A \to J(B))$ we define a partial function $U(F) : A \to S \to \mathbb{N}$. The number $U(F)(a)(x)$, if defined, is the smallest n such that

$$F^n(\bot)(a)(x) \neq \kappa_1(*).$$

Definition 3. *For a continuous mapping $F: (A \to J(B)) \to (A \to J(B))$ we define the (recursive) statement $\mathsf{rec}(F): A \to J(B)$ as:*

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

Of course, like the semantics of while earlier, rec is a least fixed point:

Proposition 3. *The above statement $\mathsf{rec}(F)$ is the least fixed point of F for continuous F.* □

Continuity of a particular F is typically easy to show; by Lemma 1 all extensions – like `&` and `&&` in Example 1 – are continuous, and it is a standard result that application, lambda abstraction, composition, and the taking of least fixed points all preserve continuity.

6 Meta-theory in PVS

The semantical basis for verifications of Java programs as sketched in Section 3 has (of course) been formalised in PVS and Isabelle/HOL, to allow actual verification of Java programs using these theorem provers. Sections 4 and 5 describe a mathematical abstraction of this semantical basis in terms of a computational monad J. Also this abstract meta-theory has also been formalised in PVS. This formalisation is not the one that is used for actual verification of Java programs, but is used only here for avoiding errors. Indeed, the definitions and proofs in this paper involve many case distinctions, and it is easy to make mistakes there, *e.g.* by accidentally skipping an option.

For those readers familiar with PVS, we show three crucial definitions. First of all, the monad (3) is defined, in a PVS theory with parameter type `A` as:

```
J : TYPE = [S -> lift[union[[S, A], [S, E]]]]
```

It involves the predefined `lift` type constructor, corresponding to $1 + (-)$, and the coproduct constructor `union` for $+$. The Cartesian product in PVS is described via square brackets `[_,_]`, and the function space via `[_->_]`. The multiplication natural transformation μ is then defined as:

```
mult : [J[J[A]] -> J[A]] =
  LAMBDA(z : J[J[A]]) :
    LAMBDA(x : S) :
      CASES z(x) OF
        bottom : bottom,
        up(w) : CASES w OF
                  inl(u) : proj_2(u)(proj_1(u)),
                  inr(v) : up(inr(v))
                ENDCASES
      ENDCASES
```

In such a way all the definitions, results and proofs in this paper have been formalised in PVS.

One of the trickiest parts in the formalisation in PVS concerns the $N(c, s)$ and $E(c, s)$ used in the definition of the while. We shall describe $N(c, s)(a)(x)$ as a set of numbers, see also [7]. It is either empty or a singleton.

```
N(c, s)(a)(x) : PRED[nat] = LAMBDA(n : nat) :
  CASES iterate(s o E2S(c), n)(a)(x) OF
    bottom : FALSE,
    up(w) : CASES w OF
              inl(u) : CASES c(proj_1(u)) OF
                         bottom : FALSE,
                         up(z) : CASES z OF
                                   inl(s) : NOT proj_2(s),
                                   inr(s) : FALSE
                                 ENDCASES
                       ENDCASES,
              inr(v) : FALSE
            ENDCASES
  ENDCASES
    AND
```

```
FORALL(m : nat) : m < n IMPLIES
  CASES iterate(s o E2S(c), m)(a)(x) OF
    bottom : FALSE,
    up(w) : CASES w OF
              inl(u) : CASES c(proj_1(u)) OF
                         bottom : FALSE,
                         up(z) : CASES z OF
                                   inl(s) : proj_2(s),
                                   inr(t) : FALSE
                                 ENDCASES
                       ENDCASES,
              inr(v) : FALSE
            ENDCASES
  ENDCASES
```

The notation `E2S(c)` is used for what we have written as $\widehat{c}$ in the beginning of Subsection 5.1.

7 Conclusions

We have discussed the monad that is at the heart of the semantics of sequential Java used in the LOOP project as the basis of mechanically assisted verification of actual Java programs. This monad is a useful notion in organizing the Java semantics, particularly when it comes to the subtleties involved with handling abnormal termination.

It is interesting to compare our approach to the one taken in the denotational semantics of sequential Java described in [1], which uses continuations to deal with exceptions and breaks. In our approach the monad J makes explicit everything involved with the control flow of programs. 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 continuation information.

A disadvantage of the continuation 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[3]. In our monadic approach the type information of the very basic type system discussed in Section 2 provides some safety: for the definitions to be well-typed we are 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.

[3] 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.

The monadic approach is less ad-hoc, in that the same basic machinery is used to deal with nontermination and abrupt termination. The monadic approach is also more general. For example, extending the semantics to cope with threads could possibly be achieved by considering a more complicated monad than J, but such a semantics of concurrent Java is still left as future work.

One crucial ingredient of Java is throwing and catching of exceptions. Appropriate functions can be defined in the current setting: for $e \in E$, take

$$1 \xrightarrow{\mathsf{throw}(e)} J(A)$$

as

$$\mathsf{throw}(e)(*) = \lambda x \in S.\, \kappa_3(x, e)$$

and for $C \subseteq E$,

$$J(A) \times J(A)^E \xrightarrow{\mathsf{try\_catch}(C)} J(A)$$

as

$$\begin{array}{rl} \mathsf{try\_catch}(C)(s,t) = \lambda x \in S.\ \mathsf{CASES}\ s(x)\ \mathsf{OF}\ \{ & \\ & \kappa_1(u) \mapsto \kappa_1(u), \\ & \kappa_2(x', a) \mapsto \kappa_2(x', a), \\ & \kappa_3(x', e) \mapsto \mathsf{IF}\ e \in C \\ & \qquad\qquad\quad \mathsf{THEN}\ t(e)(x') \\ & \qquad\qquad\quad \mathsf{ELSE}\ \kappa_3(x', e)\ \} \end{array}$$

What is slightly unsatisfactory about these definitions is that they are *ad hoc* from a mathematical perspective: it is not clear in what sense they are canonical (like composition, extension or recursion) and what basic laws they (should) satisfy. This remains to be investigated.

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*, number 1523 in Lect. Notes Comp. Sci., pages 201–240. Springer, Berlin, 1998.
2. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. Techn. Rep. CSI-R9924, Comput. Sci. Inst., Univ. of Nijmegen, 1999.
3. P. Cenciarelli. An algebraic view of program composition. In A.M. Haeberer, editor, *Algebraic Methodology and Software Technology*, number 1548 in Lect. Notes Comp. Sci., pages 325–340. Springer, Berlin, 1998.
4. M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* Cambridge Univ. Press, 1996.
5. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
6. M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. Techn. Rep. CSI-R0004, Comput. Sci. Inst., Univ. of Nijmegen, 1999.

7. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci. Springer, Berlin, 2000.
8. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs. Proceedings of the ECOOP'99 Workshop*, pages 37–44. Techn. Rep. 251, Fernuniversität Hagen, 1999.
9. B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in Lect. Notes Comp. Sci., pages 210–231. Springer, Berlin, 1996.
10. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
11. 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*, pages 329–340. ACM Press, 1998.
12. E. Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.
13. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
14. 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.
15. Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.
16. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
17. J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Semantics.* MIT Press, Cambridge, MA, 1977.