

Parametrized Exceptions

Dennis Walter, Lutz Schröder, and Till Mossakowski

BISS, Department of Computer Science, University of Bremen

Abstract. Following the paradigm of encapsulation of side effects via monads, the Java execution mechanism has been described by the so-called Java monad, incorporating essentially stateful computation and exceptions, which are heavily used in Java control flow. A technical problem that appears in this model is the fact that the return exception in Java is parametrized by the return value, so that method calls actually move between slightly different monads, depending on the type of the return value. We provide a treatment of this problem in the general framework of exception monads as introduced in earlier work by some of the authors; this framework includes generic partial and total Hoare calculi for abrupt termination. Moreover, we illustrate this framework by means of a verification of a pattern match algorithm.

Introduction

Many imperative languages allow for manipulating the control flow by means of exceptions. Particularly extensive use of this possibility is made in Java, where abnormally terminating statements are used e.g. in order to exit from loops or method calls. Exceptions therefore also play a prominent role in the design of program logics for Java.

Generally, imperative languages may be represented in standard higher order logic or in functional programming languages by encapsulating side effects as monads, a principle introduced by Moggi [9]. The Java exception mechanism has been modelled by the so-called Java monad [6], an instance of Moggi's exception monad transformer [8]. In previous work [16, 14, 15], we have developed monadic computational logics for generic side effects, and we have extended these logics with a generic treatment of exceptional termination that subsumes existing Hoare logics for abrupt termination [4, 5].

A technical problem that appears in the monadic modeling of the Java exception mechanism is the fact that the return exception in Java is parametrized by the value to be returned, so that method calls actually move between slightly different monads, depending on the type of the return value. In previous work [4], this problem has been treated by a workaround which involves storing the return value in a global variable. Here, we provide a more elegant solution in the shape of a monadic wrapper routine for method bodies that shifts the value of the return exception (which is now treated as a part of the exception, in accordance with the Java language specification [7]) into the value of the monadic computation. We also provide suitable Hoare rules for this wrapper. The generic framework

is illustrated by means of a benchmark problem appearing also in [4, 6], the verification of a pattern match algorithm, thus also providing a first example application for the calculus presented in [15].

1 Monads for Imperative Programming and Specification

Following seminal work by Moggi [9], monads are being used in both semantics and programming to formalize and encapsulate side effects in an elegant, functional way; in particular, this idea is one of the central concepts of Haskell [11]. Intuitively, a monad associates to each type A a type TA of *computations* of type A ; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \rightarrow TB$, also called a (*B-valued*) *program*. This approach abstracts away from particular notions of computation such as store, non-determinism, non-termination etc.; a surprisingly large amount of reasoning can in fact be carried out independently of the choice of such a notion.

Formally, there are three ingredients to a monad: the computation type constructor T , a lifting operation $\text{ret} : a \rightarrow T a$ for each type a , and a binding operation $_ \gg= _ : T a \rightarrow (a \rightarrow T b) \rightarrow T b$ for all types a, b . The intuition behind these operations is that $\text{ret } x$ is a computation without side-effects that just returns the value x (not to be confused with the return exceptions appearing further below), and that $p \gg= f$ executes the computation $p : T a$ and feeds the resulting values of type a into the program $f : a \rightarrow T b$; this is essentially sequential composition of statements. A *strong monad* additionally has an operation $t : a \times T b \rightarrow T (a \times b)$, called (*tensorial*) *strength*, for all types a, b . These data are governed by equational axioms requiring associativity of binding, neutrality of $\text{ret } x$ w.r.t. binding, and compatibility of the strength with binding and lifting. A slight complication is caused by the fact that programs $a \rightarrow T b$ are in practice typically partial functions; a version of the monad axioms accommodating partial functions in such a way that typical monads such as the state monad of Example 1 below are actually subsumed (cf. discussions on this topic in [3]) is given in [16].

Example 1 ([9]). Computationally relevant monads on **Set** (all monads on **Set** are strong) include

- stateful computations with possible non-termination: $TA = (S \rightarrow? (A \times S))$, where S is a fixed set of states and $_ \rightarrow? _$ denotes the partial function type;
- (finite) non-determinism: $TA = \mathcal{P}_{fin}(A)$, where \mathcal{P}_{fin} denotes the finite power set functor;
- exceptions: $TA = A + E$, where E is a fixed set of exceptions;
- interactive input: TA is the least fixed point of $\lambda\gamma. A + (U \rightarrow? \gamma)$, where U is a set of input values;
- non-deterministic stateful computations: $TA = S \rightarrow \mathcal{P}_{fin}(A \times S)$, where, again, S is a fixed set of states.

As laid out in [18] and incorporated in the Haskell syntax, monads can be used to support an imperative style of notation: terms of the form

$$\text{do } x \leftarrow p; q$$

(where x may appear in q) are taken to abbreviate $p \gg= \lambda x. q$, and nested bindings $\text{do } x \leftarrow p_1; \text{do } y \leftarrow p_2; \dots$ are denoted in the form $\text{do } x \leftarrow p_1; y \leftarrow p_2; \dots$. Sequences of bindings $x_1 \leftarrow p_1; \dots; x_n \leftarrow p_n$ will often be indicated by a bar notation $\bar{x} \leftarrow \bar{p}$ below. As indicated above, the notational coincidence with the usual sequential composition operator indeed conveys the right intuition, in particular in connection with variants of the above-mentioned state monad. Imperative control structures such as the while loop can be expressed as recursive programs in this notation; the definition of a slightly generalized loop construct *iter* where a return value is passed through the loop is shown in Fig. 1, along with a definition of the while loop by specialization to void results.

The background formalism used in Fig. 1 and in the further development is the wide-spectrum language HASCASL, which extends the standard algebraic specification language CASL by features aimed at providing support for modern functional programming languages. These features include higher order logic of partial functions, type class polymorphism in the style of Haskell, and general recursion over domains, specified in a bootstrap fashion as complete partial orders (cpo's) in much the same way as in HOLCF [12]. In the following, we briefly explain some of the less obvious syntactical features of HASCASL appearing in Fig. 1 and elsewhere; for further details, the reader is referred to [13, 17].

As indicated above, the treatment of recursion in HASCASL is via a type class *Cpo* of complete partial orders; type classes are just subsets of the syntactical universe of types. The types of total and partial continuous functions between cpo's a and b are denoted by $a \xrightarrow{\text{cont}} b$ and $a \xrightarrow{\text{cont}}? b$, respectively (in contrast to non-continuous function types $a \rightarrow b$ and $a \rightarrow? b$). Operators containing type variables of a certain type class in their type are thereby declared to be polymorphic over that class; e.g. the *iter* operation of Fig. 1 is polymorphic over a so-called cpo monad T , i.e. a monad that allows lifting cpo structures on a to cpo structures on $T a$ in such a way that the monad operations become continuous. Here, the class *CpoMonad* is, like the more general class *Monad* of monads, a *constructor class*, i.e. a subset of the syntactical universe of type constructors. The specification of the class *CpoMonad* is imported by referencing the named specification CPOMONAD (not shown here), which itself includes the specification of both cpo's and monads.

The keyword **program** invokes syntactical sugar for the fixed point operator on cpo's which allows writing programs in much the same style as e.g. in Haskell; in particular, the equation defining *iter* in Fig. 1 is an actual recursive definition rather than just a semantic equation. Program blocks in HASCASL can be automatically translated into Haskell by means of the Bremen heterogeneous tool set [10].

As indicated above, HASCASL incorporates a higher order logic of partial functions. Below, we will freely combine the monadic do-notation and higher

order formulae, using e.g. monadic computations of truth values. A more detailed explanation of the required logical framework is given in [16]. We stress that the use of HASCASL serves mainly to drive home the point that our framework can be cast in a real specification language; the results as such remain valid essentially in any suitable higher order logic of partial functions, including standard set theory and topos logic.

<pre> spec ITERATION = CPOMONAD and BOOL then vars T: CpoMonad; a: Cpo op iter: (a \xrightarrow{cont} T Bool) \xrightarrow{cont} (a $\xrightarrow{cont?}$ T a) \xrightarrow{cont} a $\xrightarrow{cont?}$ T a program iter test f x = do b \leftarrow test x if b then do y \leftarrow f x iter test f y else return x op while (b: T Bool) (p: T Unit): T Unit = iter ($\lambda x \bullet b$) ($\lambda x \bullet p$) () </pre>

Fig. 1. The iteration control structure

In [16, 14], generic computational logics have been introduced that allow reasoning about the correctness of monadic programs. This includes a Hoare logic for partial correctness, a dynamic logic, and a Hoare logic for total correctness defined via dynamic logic. While partial Hoare logic works in principle over arbitrary monads, the semantics of dynamic logic (and hence also of total Hoare logic) is introduced axiomatically and works only for sufficiently well-behaved monads, including the monads of Example 1 and excluding e.g. the continuation monad; in the positive case we say that a monad *admits dynamic logic*.

A basic concept underlying all these logics is the notion of *deterministically side-effect free (dsef)* program engendered by the underlying monad. In the terminology of [2], p is dsef if p is central in the variety of discardable and copyable programs. Intuitively, discardability means that the state may be read, but not changed (this is in contrast to *stateless* programs, i.e. programs of the form `ret a`, which additionally do not access the state), and copyability amounts to determinism.

The monadic partial Hoare calculus is concerned with Hoare triples

$$\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi\},$$

where $\bar{x} \leftarrow \bar{p}$ is a sequence of bindings for monadic programs p_i , ϕ and ψ are *formulae*, i.e. dsef computations of truth values, and ψ may mention the intermediate results x_i . The semantics of Hoare triples is defined in terms of equations between computations of truth values. In the monads of Example 1, this semantics instantiates as expected. For example, a Hoare triple $\{\phi\} x \leftarrow p \{\psi\}$ holds in the state monad iff, whenever ϕ holds in a state s , then ψ holds for x after

successful execution of p from s with result x . In the non-deterministic state-monad, ψ must be satisfied for all possible pairs of results and post-states for p . A calculus for such Hoare triples may be derived directly from the definition of the semantics; the rules of the calculus include e.g. a sequencing rule

$$\text{(seq)} \frac{\frac{\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi\}}{\{\psi\} \bar{y} \leftarrow \bar{q} \{\chi\}}}{\{\phi\} \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q} \{\chi\}}$$

and, for cpo monads, a loop rule

$$\text{(iter)} \frac{\{\phi \ x \wedge b \ x\} \ y \leftarrow p \ x \ \{\phi \ y\}}{\{\phi \ e\} \ y \leftarrow \text{iter } b \ p \ e \ \{\phi \ y \wedge \neg(b \ y)\}} \text{ (b dsef)}$$

which generalizes the usual while rule.

Hoare triples $[\phi] \bar{x} \leftarrow \bar{p} [\psi]$ for *total* correctness are, as indicated above, defined using the diamond operator of monadic dynamic logic [16] and hence can be interpreted only over monads that admit dynamic logic, which however does not seem to be an overly serious restriction. As for partial Hoare triples, one can prove the usual set of rules from the definition of the semantics, including e.g. a rule for total correctness of loops which specializes to the usual total while rule.

2 Exception Monads and the Java Monad

In modern imperative languages, exceptions are often used as a means of manipulating the control flow, replacing explicit jumps. A maybe somewhat extreme example is Java, where common control statements such as **break**, **continue**, and **return** terminate abnormally, and where e.g. bodies of methods with non-void result type are in fact expressly forbidden to terminate normally. Hoare calculi for Java as developed e.g. in [4, 5] therefore need to accommodate correctness assertions for abnormal termination.

In the monadic setting, exceptions are modelled by Moggi's *exception monad transformer* [8] which, given a set E of exceptions, transforms a monad R into the associated exception monad $Ex \ E \ R = R \ (_ + E)$ — i.e. $Ex \ E \ R$ models the side effects of R , extended by exceptions in E . A simple example is the *Java monad* [6] $Ex \ E \ ST$, where ST is the state monad of Example 1. In principle, the generic computational logics described in the previous section do apply to exception monads; in particular, $Ex \ E \ R$ admits dynamic logic if R does. However, since monadic computational logic as recalled in the previous section treats exceptional termination like non-termination in the sense that one has $\{\} p \{\perp\}$ if p throws an exception, the generic framework, like the concrete Hoare calculi of [4, 5], needs to be provided with an explicit treatment of abnormal termination.

Such a generic framework for exception monads has been introduced in [15]. It is obtained by first giving an equational characterization of exception monads in the above sense, and then combining this with the existing monadic Hoare logics

as explained in Sect. 1. The equational description of an exception monad T is based on operations $catch : T a \rightarrow T (a + E)$ and $raise : E \rightarrow T a$ for each type a . Here, $raise e$ throws an exception, thus freezing the state (so that subsequent bindings are skipped until the exception is caught), and $catch p$ behaves like p if p terminates normally, and otherwise returns a thrown exception and unfreezes the state, i.e. resumes normal execution of statements. If $T = Ex E R$, then $catch p$, which for $p : Ex E R a$ is of type $Ex E R (a + E) = R ((a + E) + E)$, may be expressed in the do-notation for R as

$$\text{do } x \leftarrow p; \text{ case } x \text{ of } \text{inl } y \rightarrow \text{ret } (\text{inl } (\text{inl } y)) \mid \text{inr } e \rightarrow \text{ret } (\text{inl } (\text{inr } e)).$$

It is convenient to let $catch \bar{x} \leftarrow \bar{p}$ abbreviate $catch (\text{do } \bar{x} \leftarrow \bar{p}; \text{ret } \bar{x})$.

One can then define Hoare assertions that simultaneously cover normal and abnormal termination. A *partial extended Hoare assertion* $\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi \parallel S\}$ abbreviates

$$\{\phi\} y \leftarrow (catch \bar{x} \leftarrow \bar{p}) \{\text{case } y \text{ of } \text{inl } \bar{x} \rightarrow \psi \mid \text{inr } e \rightarrow S e\},$$

thus stating that, if the program sequence $\bar{x} \leftarrow \bar{p}$ is executed in a state that satisfies ϕ , then if the execution terminates normally, the post-state and the result \bar{x} satisfy ψ , and if the execution terminates abnormally with exception e , the post-state satisfies $S e$ (in particular note that $S e$ is a stateful formula, although as usual required to be dsef). The conditions ψ and S are referred to as the *normal* and the *abnormal postcondition*, respectively. Similarly, *total extended Hoare assertions* $[\phi] \bar{x} \leftarrow \bar{p} [\psi \parallel S]$ are defined using $catch$ and standard total Hoare triples; the meaning of a total assertion is thus the conjunction of the partial assertion and termination of $catch \bar{x} \leftarrow \bar{p}$. A calculus for extended Hoare assertions is then easily derived from the standard Hoare calculus and the equational description of exception monads. The rules for partial extended Hoare assertions are shown in Figure 2, with the general fixed point rule (Y) of [15] replaced by its specialization to *iter* (which then gives rise to a *while* rule by further specialization). The rules for total extended Hoare assertions are largely the same, except for the loop rule, which becomes

$$\text{(iter-total)} \frac{\begin{array}{c} \dots < \dots : \text{Pred } (c \times c) \text{ is well-founded} \\ [\phi x \wedge b x \wedge (t x = z)] y \leftarrow p x [\phi y \wedge (t y < z) \parallel S] \end{array}}{[\phi e] y \leftarrow \text{iter } b p e [\phi y \wedge \neg(b y) \parallel S]}$$

(subject to the side condition that $t x : T c$ is dsef for all $x : a$). As explained in [15], this rule can be specialized to the *total exception while rule*

$$\frac{\begin{array}{c} [\phi \wedge b] p [\top \parallel \top] \\ \{\phi \wedge b \wedge t = z\} p \{\phi \wedge b \wedge t < z \parallel \top\} \\ \{\phi \wedge b\} p \{\top \parallel S\} \end{array}}{[\phi \wedge b] \text{while } b p [\perp \parallel S]}$$

Further below, we will use a slight variant of this rule, where the well-founded relation $<$ lives only on a subtype of c and the invariant ϕ guarantees that results of t are always in this subtype.

$$\begin{array}{c}
\text{(seq)} \frac{\frac{\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi \parallel S\}}{\{\psi\} \bar{y} \leftarrow \bar{q} \{\chi \parallel S\}}}{\{\phi\} \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q} \{\chi \parallel S\}} \quad \text{(ctr)} \frac{\{\phi\} \dots; x \leftarrow p; y \leftarrow q; \bar{z} \leftarrow \bar{r} \{\psi \parallel S\}}{x \notin FV(\bar{r}) \cup FV(\psi)} \\
\{\phi\} \dots; y \leftarrow (\text{do } x \leftarrow p; q); \dots \{\psi \parallel S\} \\
\\
\text{(conj)} \frac{\frac{\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi_1 \parallel S_1\}}{\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi_2 \parallel S_2\}}}{\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi_1 \wedge \psi_2 \parallel S_1 \wedge S_2\}} \quad \text{(disj)} \frac{\frac{\{\phi_1\} \bar{x} \leftarrow \bar{p} \{\psi \parallel S\}}{\{\phi_2\} \bar{x} \leftarrow \bar{p} \{\psi \parallel S\}}}{\{\phi_1 \vee \phi_2\} \bar{x} \leftarrow \bar{p} \{\psi \parallel S\}} \\
\\
\text{(wk)} \frac{\frac{\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi \parallel S\}}{\phi' \Rightarrow \phi \quad \psi \Rightarrow \psi'}}{S \Rightarrow S'}}{\{\phi'\} \bar{x} \leftarrow \bar{p} \{\psi' \parallel S'\}} \quad \text{(stateless)} \frac{}{\{\text{ret } \phi\} q \{\text{ret } \phi \parallel \lambda e. \text{ret } \phi\}} \\
\\
\text{(dsef)} \frac{p \text{ dsef}}{\{\phi\} x \leftarrow p \{\phi \wedge x = p \parallel \perp\}} \quad \text{(catch)} \frac{\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi[inl \bar{x}/y] \parallel \lambda e. \psi[inr e/y]\}}{\{\phi\} y \leftarrow (\text{catch } \bar{x} \leftarrow \bar{p}) \{\psi \parallel \perp\}} \\
\\
\text{(raise)} \frac{}{\{\phi\} \text{raise } e_0 \{\perp \parallel \lambda e. (\phi \wedge e = e_0)\}} \quad \text{(if)} \frac{\frac{\{\phi \wedge b\} x \leftarrow p \{\psi \parallel S\}}{\{\phi \wedge \neg b\} x \leftarrow q \{\psi \parallel S\}}}{\{\phi\} x \leftarrow \text{if } b \text{ then } p \text{ else } q \{\psi \parallel S\}} \\
\\
\text{(iter)} \frac{\{\phi \wedge b x\} y \leftarrow p x \{\phi y \parallel S\}}{\{\phi e\} y \leftarrow \text{iter } b p e \{\phi y \wedge \neg(b y) \parallel S\}}
\end{array}$$

Fig. 2. The generic Hoare calculus for partial exception correctness

It is shown in [15] that these generic Hoare calculi subsume the calculi of [4, 5] w.r.t. the treatment of exceptions (the work of [4, 5] covers also aspects of the Java class mechanism, which is not considered here).

Remark 2. The very simple exception mechanism laid out above can be used to capture also the more involved aspects of Java’s treatment of exceptions. For instance, the monadic approach is also suitable for the modeling of side-effecting expressions. In the encoding, an expression needs to be decomposed into a sequence of bindings in a `do`-expression, where each binding corresponds to an application of a method. Hoare rules that actually work on unencoded expressions are easily designed using this observation, given a fixed order of evaluation for subexpressions.

Additionally, the following program *tryFinally* models Java’s `try` statement with a `finally` block attached. It guarantees execution of the program *q* representing the finally block even in case of abnormal termination of the try block program *p*. It terminates normally if both *p* and *q* do. Otherwise, it raises the exception of *p* or *q* with *q*’s exceptions overwriting those of *p*:

$$\text{tryFinally } p \ q = \text{do } x \leftarrow \text{catch } p \\ \text{case } x \text{ of } \text{inl } _ \rightarrow q \mid \text{inr } e \rightarrow \text{do } q; \text{raise } e$$

(i.e. *tryFinally p q* corresponds to `try p finally q`; additional `catch` clauses can be coded in the obvious way). A corresponding Hoare rule is easily derived from existing rules of the calculus (cf. Fig. 2), making supplementary use of a rule for the case construct. The total Hoare rule shown below (the partial rule is analogous) in particular captures the fact that exceptions in the finally block dominate those of the try block. Moreover, it subsumes a rule stating that exceptions in *p* propagate beyond *q* supposing *q* itself does not raise exceptions.

$$\text{(try-finally)} \frac{\begin{array}{c} [\phi] p \ [\chi \parallel R] \\ [\chi] q \ [\psi \parallel S] \\ [Re] q \ [Se \parallel S] \end{array}}{[\phi] \text{tryFinally } p \ q \ [\psi \parallel S]} \text{ (} e \text{ fresh)}$$

3 Parametrized Exceptions and Java Return Values

As indicated in the introduction, the translation of Java programs into the monadic framework faces the following technical problem. In Java, the only admissible way for a method to return a value is via a `return` statement, with the returned value as parameter. The `return` statement terminates abnormally, essentially raising an exception marked as a return exception and *containing* the return value. The rest of the method body is then skipped; the return exception is implicitly caught at the end of the method body, where the exceptional return value is turned into a normal result and normal execution is resumed. In the monadic framework, this means that the body *p* of a method *m* with type *a* of return values is of the type

$$p : Ex (E \ a) \ R \ b,$$

where $E a$ is a parametrized type of exceptions, with return exceptions carrying values of type a , and R is the underlying monad; the type b of ‘normal’ results does not really matter, since the method body is explicitly forbidden to terminate normally [7]. The standard *catch* function could be used to turn this into $catch\ p : Ex\ (E\ a)\ R\ (b + E\ a)$; however, if the method call to m took place in the body of a further method with result type c , use of *catch p* would still lead to a type error since it is a computation in $Ex\ (E\ a)\ R$ rather than $Ex\ (E\ c)\ R$.

Possibly for this reason, the translation of method calls in the LOOP tool as described e.g. in [4] slightly deviates from the above procedure: return exceptions are treated as unparametrized, so that there is a monomorphic type E of exceptions. In order to pass the return value of a method to the caller, one then needs to side-step the exception mechanism, creating instead a new global variable in which the return value is stored at the time of execution of the **return** statement and from which it is later retrieved by the wrapper function of the method call. It is clear that this solution is not entirely satisfactory. We will now propose an alternative solution which conforms to the Java Language Specification.

We will work with a polymorphic datatype

$$E\ a = MRet\ a \mid DropOff \mid \dots$$

of exceptions, parametrized by the type a of return values. Here, $MRet$ is the return exception carrying the return value, and *DropOff* (‘dropped off end of method body’) is a special exception to be raised when a non-void method terminates normally (i.e. never, since this case should according to [7] be caught at compile time — however, we will still need to insert some value into the corresponding case statements); the unmentioned further exceptions do not depend on a .

In our calculus, Java **return** statements are translated into the throwing of return exceptions, i.e. statements of the form *raise* ($MRet\ x$), abbreviated by *mret x*. Similar to [4], every method body is protected by a wrapper function *mbody*, so that for each method m with body p one has $m = mbody\ p$. This wrapper function turns the abnormal state caused by a return exception back into a normal one; but in contrast to the existing *catch* operation it additionally allows shifting the type of return exceptions. This is to say that *mbody* needs to have the polymorphic type

$$mbody : Ex\ (E\ a)\ R\ b \rightarrow Ex\ (E\ c)\ R\ a$$

for all types a, b, c ; the instantiation of c is then determined by the context of the monadic computation in which the method call appears.

In order to give a generic definition of *mbody*, we recall that, while we will usually want to write monadic programs directly in $Ex\ (E\ a)\ R$, we can also exploit our knowledge that $Ex\ (E\ a)\ R = R\ (- + E\ a)$, and program in the

monadic notation for R . Thus, we can write

$$\begin{aligned}
mbody\ p &= do \\
&\quad x \leftarrow p; \\
&\quad case\ x\ of \\
&\quad\quad inl\ _ \rightarrow ret\ (inr\ DropOff) \\
&\quad\quad | inr\ e \rightarrow case\ e\ of \\
&\quad\quad\quad MRet\ v \rightarrow ret\ (inl\ v) \\
&\quad\quad\quad | _ \rightarrow ret\ (inr\ e)
\end{aligned}$$

(type checking e.g. in Isabelle or Haskell will detect that this is the do-notation in R rather than in $Ex\ (E\ a)\ R$).

In order to conduct proofs about programs involving $mbody$, we need to encapsulate its properties in suitable Hoare rules at the level of $Ex\ (E\ a)\ R$, i.e. in the Hoare calculi for abrupt termination. In both the total and the partial calculus, a single Hoare rule suffices to prove properties of programs that obey the above-mentioned restrictions, i.e. where method bodies never return values normally, but always via the $mret$ statement. The partial Hoare rule is

$$(mbody) \frac{\{\phi\} x \leftarrow p \ \{\perp \parallel \lambda e. case\ e\ of\ MRet\ y \rightarrow \psi \mid e \rightarrow S\ e\}}{\{\phi\} y \leftarrow mbody\ p \ \{\psi \parallel S\}},$$

the total version is analogous (the only additional statement being that $mbody\ p$ terminates, normally or abruptly, if p terminates). Given the definition of $mbody$ above, the proof of these rules in the monadic Hoare calculus (without abrupt termination) for R is straightforward, recalling the proper definition of the binding operation for $Ex\ (E\ a)\ R$ in the do-notation for R .

Note that from these rules one can infer further intuitively expected properties. E.g. by choosing S such that $S\ (MRet\ x) = \perp$ for each x , one obtains that $mbody\ p$ does not throw or let pass any return exceptions. Furthermore, one concludes that $mbody\ p$ only raises a $DropOff$ exception —indicating that p terminated normally— if either p indeed terminates normally or p itself raised a $DropOff$ exception, by choosing S such that $S\ DropOff = \perp$. Further illustration of the use of the rules is given in the example proof in the next section.

4 Verification of a Pattern Match Algorithm

To give an idea of how the calculus described above is applied in the verification of programs exploiting abnormal termination mechanisms, we will now prove the correctness of a pattern match algorithm. This algorithm has already been used as an example for the application of the calculi of [4, 5]; the main point to be made here is that a concrete algorithm of this kind can indeed be verified in our generic framework.

The algorithm is implemented in an exception monad with dynamic references (implemented in our concrete algorithm as natural numbers). One therefore has to axiomatize additional operations on the monad (apart from ret and

$\gg=$); the corresponding HASCASL specification is shown in Figure 3. The notation should be largely self-explanatory. The type *Logical* is HASCASL's built-in type of truth values. The imported specification EXCEPTIONMONAD defines the exception monad transformer *Ex*, which is used to generate a reference monad *RE* with exceptions over a loosely specified base monad *R*. This specification, in turn, imports the specification CPOMONAD (cf. Section 1); the imported symbols include the higher order type constructor *D* which, given a monad *T* and a type *a*, extracts the type *D T a* of deterministically side-effect free *a*-valued *T*-computations. The specification in Figure 3 extends the axiomatization of the dynamic reference monad in terms of 'normal' Hoare triples given in [14] by abnormal postconditions, which in most cases are \perp , asserting that the corresponding operations do not raise exceptions. An exception is the rule (new-distinct), which states that subsequent creation of references, with an arbitrary program *p* (which may raise exceptions) executed in between, produces distinct references.

<pre> spec EXCREFERENCE = EXCEPTIONMONAD then var <i>a</i>: <i>Cpo</i> types <i>R</i>: <i>CpoMonad</i>; <i>Ref a</i>, <i>E</i>: <i>Flatcpo</i>; type <i>RE</i> := <i>Ex E R</i> ops <i>read</i>: <i>Ref a</i> \xrightarrow{cont} <i>D RE a</i>; <i>.. := ..</i>: <i>Ref a</i> \xrightarrow{cont} <i>a</i> \xrightarrow{cont} <i>RE Unit</i> forall <i>x, y</i>: <i>a</i>; <i>r, s</i>: <i>Ref a</i> • [] <i>r</i> := <i>x</i> [<i>x</i> = *<i>r</i> \perp] <i>%(read-write)%</i> • [<i>x</i> = *<i>r</i> \wedge \neg<i>r</i> = <i>s</i>] <i>s</i> := <i>y</i> [<i>x</i> = *<i>r</i> \perp] <i>%(read-write-other)%</i> </pre>
<pre> spec DYNAMICEXCREFERENCE = REFERENCE then var <i>a, b</i>: <i>Type</i> op <i>new</i>: <i>a</i> \xrightarrow{cont} <i>RE (Ref a)</i> forall <i>x, y</i>: <i>a</i>; <i>t</i>: <i>Ref a</i>; <i>p</i>: <i>Ref a</i> \rightarrow <i>RE b</i>; <i>P</i>: <i>D RE Logical</i> • [] <i>r</i> \leftarrow <i>new x</i> [<i>x</i> = *<i>r</i> \perp] <i>%(read-new)%</i> • [<i>y</i> = *<i>t</i>] <i>r</i> \leftarrow <i>new x</i> [<i>y</i> = *<i>t</i> \vee <i>t</i> = <i>r</i> \perp] <i>%(read-new-other)%</i> • [<i>P</i>] <i>r</i> \leftarrow <i>new x</i>; <i>p r</i> [\top \top] \Rightarrow [<i>P</i>] <i>r</i> \leftarrow <i>new x</i>; <i>p r</i>; <i>s</i> \leftarrow <i>new y</i> [\neg<i>r</i> = <i>s</i> \top] <i>%(new-distinct)%</i> </pre>

Fig. 3. Specification of the reference and the dynamic reference monad

A Haskell implementation of the pattern match algorithm (which does not really look all that different from the corresponding executable HASCASL specification) is shown in Figure 4. For syntactic reasons, the function *read* replaces the *-notation of Fig. 3. The infrastructure functions *mbody*, *mret*, and *raise* are implemented elsewhere as described above.

We prove total correctness of the algorithm generically, i.e. without further assumptions on the underlying monad other than the axioms of Figure 3. For convenience, we make use of *existential equality* $\stackrel{e}{=}$ e.g. when comparing elements of lists. For instance, $a!!i \stackrel{e}{=} b!!i$ means that $a!!i$ and $b!!i$ are *defined* and equal, where $(!!) :: List\ a \rightarrow Int \rightarrow a$ is the indexing function for the list datatype, with $a!!i$ defined only for $0 \leq i < len\ a$.

```

pmatch base pat = mbody (
  do r <- new 0
     s <- new 0
     while (ret  $\top$ )
       (do u <- read r
          v <- read s
          if u == len pat
            then mret v
            else if v + u == len base
              then raise PatternNotFound
              else if base!!(v+u) == pat!!u
                then r := (u+1)
                else do s := (v+1); r := 0)
)

```

Fig. 4. Haskell implementation of the pattern match algorithm

For the actual method body p , i.e. the argument of `mbody` in Figure 4, we claim that it terminates abnormally, raising either a return exception carrying as its value an index x that is the starting position of the first occurrence of the pattern in the base string or a failure exception indicating that there is no occurrence of the pattern in the base string:

$$\begin{array}{l}
\boxed{p \ [\perp \ \|\ \lambda e. \text{case } e \text{ of} \\
\quad MRet\ i \rightarrow MPOS\ i \wedge \forall j. MPOS\ j \Rightarrow i \leq j \\
\quad | PatternNotFound \rightarrow \neg \exists i. MPOS\ i \\
\quad | _ \rightarrow \perp]}
\end{array} \quad (1)$$

The abnormal postcondition above will be denoted by *POST* below. Here, *MPOS* i states that the pattern is matched at position i in the base string:

$$MPOS\ i \equiv \forall j. 0 \leq j < len\ pat \Rightarrow base!!(i+j) \stackrel{e}{=} pat!!j.$$

In order to apply the total exception while rule (cf. Sect. 2), we need to provide a loop invariant *INV* and a termination measure t . Putting

$$\begin{array}{l}
INV \equiv (\forall i. 0 \leq i < *r \Rightarrow base!!(*s+i) \stackrel{e}{=} pat!!i) \wedge \\
\quad \forall i. MPOS\ i \Rightarrow *s \leq i
\end{array}$$

5 Conclusion

The principle of encapsulation of side effects in monads can be applied to model the imperative aspects of realistic languages such as Java; in particular, the Java exception mechanism is accurately captured by the so-called Java monad. Generic program logics including partial and total Hoare logics can be formulated largely independently of the nature of specific side-effects, i.e. monads [14, 16]; specific Hoare logics for abnormal termination introduced as part of verification support frameworks for Java are also subsumed by generic logics [15].

Here, we have illustrated this principle by means of a ‘benchmark’ verification of a pattern match algorithm previously used also as a test case for existing specific Hoare logics, implemented in a loosely specified dynamic reference monad. The example has shown that the framework of [15], for which no example application had so far been provided, is able to deal with realistic examples making extensive use of abrupt termination.

A general technical problem with the monadic modelling of the Java termination mechanism that came up in the verification process is that Java return exceptions are of a polymorphic type, parametrized over the type of the return value, so that the Java monad must in fact be regarded as a ‘polymorphic monad’ — i.e. each method body is executed in the instance of the Java monad determined by its result type. In order to deal with this polymorphism, we have designed a generalized catch function to be implicitly wrapped around method bodies in the same style as in the translation implemented in the LOOP tool [4]. This wrapper function shifts return exceptions into regular monadic return values and converts the resulting computation to fit the ambient monad; this solution improves on the previous approach, which consisted in bypassing the exception mechanism by storing return values in global variables [4]. There is a natural Hoare rule for wrapped method bodies, so that method calls can be dealt with in the generic verification framework without further problems.

This work forms part of an ongoing effort to adapt the wide-spectrum language HASCASL to the specification of object-oriented programs, in particular in Java. Open problems include the modelling of the Java class mechanism in HASCASL and logical support for concurrency. There are indications that the latter may be integrated in a monadic framework by means of continuations [1]. This motivates the search for a program logic for the continuation monad, to which by the results of [16] the existing generic computational logics are not usefully applicable.

Acknowledgements

This work forms part of the DFG-funded project HasCASL2 (KR 1191/7-2).

References

- [1] K. Claessen, *A poor man’s concurrency monad*, J. Funct. Programming **9** (1999), 313–323.

- [2] C. Führtmann, *Varieties of effects*, Foundations of Software Science and Computation Structures, LNCS, vol. 2303, Springer, 2002, pp. 144–158.
- [3] *The Haskell mailing list*, <http://www.haskell.org/maillinglist.html>, 2002.
- [4] M. Huisman and B. Jacobs, *Java program verification via a Hoare logic with abrupt termination*, Fundamental Approaches to Software Engineering, LNCS, vol. 1783, Springer, 2000, pp. 284–303.
- [5] B. Jacobs and E. Poll, *A logic for the Java Modeling Language JML*, Fundamental Approaches to Software Engineering, LNCS, vol. 2029, Springer, 2001, pp. 284–299.
- [6] ———, *Coalgebras and Monads in the Semantics of Java*, Theoret. Comput. Sci. **291** (2003), 329–349.
- [7] B. Joy, G. Steele, J. Gosling, and G. Bracha, *The Java language specification*, Addison-Wesley, 2000.
- [8] E. Moggi, *An abstract view of programming languages*, Tech. Report ECS-LFCS-90-113, Univ. of Edinburgh, 1990.
- [9] ———, *Notions of computation and monads*, Inform. and Comput. **93** (1991), 55–92.
- [10] T. Mossakowski, *Heterogeneous specification and the heterogeneous tool set*, Habilitation thesis, University of Bremen, 2005.
- [11] S. Peyton-Jones (ed.), *Haskell 98 language and libraries — the revised report*, Cambridge, 2003, also: J. Funct. Programming **13** (2003).
- [12] F. Regensburger, *HOLCF: Higher order logic of computable functions*, Theorem Proving in Higher Order Logics, LNCS, vol. 971, 1995, pp. 293–307.
- [13] L. Schröder and T. Mossakowski, *HASCASL: Towards integrated specification and development of functional programs*, Algebraic Methodology and Software Technology, LNCS, vol. 2422, Springer, 2002, pp. 99–116.
- [14] ———, *Monad-independent Hoare logic in HASCASL*, Fundamental Aspects of Software Engineering, LNCS, vol. 2621, 2003, pp. 261–277.
- [15] L. Schröder and T. Mossakowski, *Generic exception handling and the Java monad*, Algebraic Methodology and Software Technology, LNCS, vol. 3116, Springer, 2004, pp. 443–459.
- [16] ———, *Monad-independent dynamic logic in HASCASL*, J. Logic Comput. **14** (2004), 571–619.
- [17] L. Schröder, T. Mossakowski, and C. Maeder, *HASCASL – Integrated functional specification and programming. Language summary*, available at http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HasCASL
- [18] Philip Wadler, *How to declare an imperative*, ACM Computing Surveys **29** (1997), 240–263.