

Generic Exception Handling and the Java Monad

Lutz Schröder and Till Mossakowski

BISS, Department of Computer Science, University of Bremen

Abstract. We develop an equational definition of exception monads that characterizes Moggi’s exception monad transformer. This axiomatization is then used to define an extension of previously described monad-independent computational logics by abnormal termination. Instantiating this generic formalism with the Java monad used in the LOOP project yields in particular the known Hoare calculi with abnormal termination and JML’s method specifications; this opens up the possibility of extending these formalisms by hitherto missing computational features such as I/O and nondeterminism.

Introduction

In the course of efforts to provide proof support for the verification of Java programs, the classical Hoare calculus [4] has been extended to encompass exception handling in Java [5, 7, 8, 26], the main challenge being to deal with poststates of abruptly terminating statements. Exceptions in Java are part of a monad for Java [9], following the paradigm of encapsulation of side effects via monads [12]. Thus, the question arises whether extended Hoare calculi for exceptions can be developed abstractly over any monad with exceptions. We answer this question positively by first characterizing Moggi’s exception monad transformer by an equational theory based on a categorical formulation, and then extending our previous work about monad-independent Hoare calculi [21, 23] to calculi for exception monads that take into account both normal and abrupt termination. The advantage of such an approach is that it is not bound to a specific combination of computational effects like the Java monad. Moreover, most of the rules of the calculus come essentially for free by just adapting the normal monad independent Hoare rules [21, 23] using the equational description of exception monads.

As the background formalism for these concepts, we use the logic of HASCASL, a higher-order language for functional specification and programming, which is basically the internal language of partial cartesian closed categories (a generalization of toposes). The paper is organized as follows. In Sections 1, 2 and 3, we recall the HASCASL logic, monads and the logic for monads built on top of HASCASL. Section 4 axiomatizes exception monads and proves that they characterize Moggi’s exception monad transformer, and Section 5 introduces the Hoare logic for exception monads.

1 HASCASL

HASCASL is a higher-order extension of CASL [2], featuring higher-order functions in the style of Moggi’s partial λ -calculus [10], type constructors, type classes and constructor classes (for details, see [22, 24]); general recursion is specified on top of this in the style of HOLCF. The semantics of a HASCASL specification is the class of its (set-theoretic) *intensional Henkin models*: a function type need not contain all set-theoretic functions, and two functions that yield the same value on every input need not be equal.

A consequence of the intensional semantics is the presence of an intuitionistic *internal logic* that lives within λ -terms, defined in terms of equality [22]. There is built-in syntactical sugar for the internal logic, invoked by means of the keyword **internal** which signifies that formulas in the following block are to be understood as formulas of the internal logic.

Categorically speaking, HASCASL’s Henkin models correspond to models in partial cartesian-closed categories (pccc’s) [20], a generalization of toposes [1]. Basic HASCASL can be seen as syntactic sugar over the internal language of a pccc, i.e. essentially intuitionistic higher order logic of partial functions.

2 Monads for computations

On the basis of Moggi’s seminal work [12], monads are being used in both semantics and programming to formalize and encapsulate side effects in an elegant, functional way. 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.

More formally, a monad on a given category \mathbf{C} can be presented as a *Kleisli triple* $\mathbb{T} = (T, \eta, -^*)$, where $T : \text{Ob } \mathbf{C} \rightarrow \text{Ob } \mathbf{C}$ is a function, the *unit* η is a family of morphisms $\eta_A : A \rightarrow TA$, and $-^*$ assigns to each morphism $f : A \rightarrow TB$ a morphism $f^* : TA \rightarrow TB$ such that

$$\eta_A^* = id_{TA}, \quad f^* \eta_A = f, \quad \text{and} \quad g^* f^* = (g^* f)^*.$$

This description is equivalent to the more familiar one via an endofunctor T with unit η and a multiplication μ [1]. A monad defines its *Kleisli category*, which has the same objects as \mathbf{C} and ‘functions with side effects’ $f : A \rightarrow TB$ as morphisms from A to B ; the Kleisli composite of two such functions g and f is just $g^* f$.

In order to support a language with finitary operations and multi-variable contexts (see below), one needs a further technical requirement: a monad is called *strong* if it is equipped with a natural transformation

$$t_{A,B} : A \times TB \rightarrow T(A \times B)$$

called *strength*, subject to certain coherence conditions.

Example 1 ([12]). 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;

Reasoning about a category **C** equipped with a strong monad is greatly facilitated by the fact that proofs can be conducted in an *internal language* [12]. The crucial features of this language are

- A type operator T , where, as above, TA contains computations of type A ;
- a polymorphic operator $\text{ret} : A \rightarrow TA$ corresponding to the unit;
- a binding construct, which we denote in Haskell’s do style instead of by let : terms of the form

$$\text{do } x \leftarrow e_1; e_2$$

are interpreted by means of the tensorial strength and Kleisli composition [12] — e.g. if the ambient context is $y : A$ and e_1 is B -valued, then the interpretation $\llbracket \text{do } x \leftarrow e_1; e_2 \rrbracket$ is $\llbracket e_2 \rrbracket^* \circ t_{A,B} \circ \llbracket (y, e_1) \rrbracket$. Intuitively, $\text{do } x \leftarrow e_1; e_2$ computes e_1 and passes the results on to e_2 . Nested do expressions like $\text{do } x \leftarrow e_1; \text{do } y \leftarrow e_2; \dots$ may also be denoted $\text{do } x \leftarrow e_1; y \leftarrow e_2; \dots$. Repeated nestings such as $\text{do } x_1 \leftarrow e_1, \dots, x_n \leftarrow e_n; e$ are somewhat inaccurately denoted in the form $\text{do } \bar{x} \leftarrow \bar{e}; e$. Sequences of the form $\bar{x} \leftarrow \bar{e}$ are called *program sequences*. Variables x_i that are not used later on may be omitted from the notation.

This language (with further term formation rules and a deduction system) can be used both in order to define morphisms in **C** and in order to prove equalities between them [12]. Indeed, the theory of exception monads presented in Section 4 is formulated in this internal language, over an arbitrary category with equalizers. Only in the context of computational *logic* (e.g. the Hoare calculus introduced in the next section), one needs the framework of a partial cartesian closed category (and its internal language, phrased in HASCASL).

Given a monad, one can generically define control structures such as a while loop (see for example [15]). Such definitions require general recursion, which is realized in HASCASL by means of fixed point recursion on cpos , with the associated fixed point operator on continuous endofunctions denoted by Y [22]. Thus, one has to restrict to monads, called *cpo-monads*, that allow lifting a cpo structure on A to a cpo structure on the type TA of computations in such a way

that the monad operations become continuous. The (executable) specification of an iteration construct is shown in Figure 1. The specification imports named specifications `BOOL` of the booleans and `CPOMONAD` of cpo-monads; $_ \xrightarrow{c} _$ and $_ \xrightarrow{c} _$ are the type constructors for the partial and total continuous function space, respectively. The iteration construct behaves like a while loop, except that it additionally passes a result value through the iterations. The while loop is just iteration ignoring the result value.

```

spec ITERATION = CPOMONAD and BOOL then
vars   $m : CpoMonad; a : Cpo$ 
op    $iter : (a \xrightarrow{c} m Bool) \xrightarrow{c} (a \xrightarrow{c} m a) \xrightarrow{c} a \xrightarrow{c} m a$ 
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    $while(b : m Bool)(p : m Unit) : m Unit = iter\ (\lambda x \bullet b)\ (\lambda x \bullet p)\ ()$ 

```

Fig. 1. The iteration control structure

3 Monad-independent Computational Logic

We now recall notions of side-effect freeness in monads [21, 25] and the recently developed monad-independent computational logics [23, 21]. Throughout, \mathbb{T} will denote a strong monad.

Like traditional Hoare logic, the monad-independent Hoare calculus is concerned with proving *Hoare triples* consisting of a stateful expression together with a pre- and a postcondition. The pre- and postconditions are required to ‘leave the state unchanged’ in a suitable sense in order to guarantee composability of Hoare triples; they may, however, read the state.

Definition 2. A program p is called *discardable* if

$$(do\ y \leftarrow p; ret\ *) = ret\ *,$$

where $*$ is the unique element of the unit type.

For example, a program p is discardable in the state monad iff p terminates and does not change the state.

A program p is called *stateless* if it factors through η , i.e. if it is just a value inserted into the monad — otherwise, it is called *stateful*. E.g. in the state monad, p is stateless iff it neither changes nor reads the state. Stateless programs are discardable, but not vice versa.

In order to define the semantics of Hoare triples below, we introduce *global dynamic judgements* of the form $[\bar{x} \leftarrow \bar{p}] \phi$, which intuitively state that ϕ holds after execution of the program sequence $\bar{x} \leftarrow \bar{p}$, where ϕ is a stateless formula (i.e. $\phi : \Omega$, where Ω is the type of truth values). Formally, $[\bar{x} \leftarrow \bar{p}] \phi$ abbreviates

$$(\text{do } \bar{x} \leftarrow \bar{p}; \text{ret}(\bar{x}, \phi)) = \text{do } \bar{x} \leftarrow \bar{p}; \text{ret}(\bar{x}, \top).$$

Definition 3. A program p is *copyable* if

$$(\text{do } x \leftarrow p; y \leftarrow p; \text{ret}(x, y)) = \text{do } x \leftarrow p; \text{ret}(x, x).$$

A program p *commutes* with a program q if

$$(\text{do } x \leftarrow p; y \leftarrow q; \text{ret}(x, y)) = \text{do } y \leftarrow q; x \leftarrow p; \text{ret}(x, y).$$

A discardable and copyable program is called *deterministically side-effect free* (dsef) if it commutes with all (equivalently: all Ω -valued) discardable copyable programs. For a type A , the subtype of TA consisting of the dsef computations is denoted DA .

Dsef programs are syntactically treated like stateless values; their properties guarantee that arising ambiguities correspond to actual equalities. A discardable program p is copyable iff it is deterministic in the sense that

$$[x \leftarrow p; y \leftarrow p] (x = y).$$

Stateless programs are dsef. In the monads of Example 1, all discardable programs are dsef, with the exception of the monads where non-determinism is involved. In these cases, a discardable program is dsef iff it is deterministic.

Definition 4. A *Hoare triple for partial correctness*, written $\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi\}$, consists of a program sequence $\bar{x} \leftarrow \bar{p}$, a precondition $\phi : D\Omega$, and a postcondition $\psi : D\Omega$ (which may contain \bar{x}). This abbreviates the formula

$$[a \leftarrow \phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \psi] a \Rightarrow b.$$

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 .

Monad-independent *dynamic logic* is used in order to also capture Hoare triples for *total correctness* [21]. Dynamic logic allows nesting modal operators of the nature ‘after execution of p ’ and the usual connectives of first order logic. This means informally that the state is changed according to the effect of p within the scope of the modal operator, but is ‘restored’ outside that scope. E.g., in a dynamic logic formula such as

$$[p] \phi \Longrightarrow [q] \psi,$$

the subformulas $[p] \phi$ and $[q] \psi$ are evaluated in the same state, while ϕ and ψ are evaluated in modified states.

Definition 5. A *formula* (of dynamic logic) is a term $\varphi : D\Omega$. The formula ϕ is *valid* if $\varphi = \text{do } x \leftarrow \varphi; \text{ret } \top$.

The usual logical connectives are defined using the do-notation. The question is now whether $D\Omega$ has enough structure to allow also the interpretation of the diamond and box operators $\langle p \rangle$ and $[p]$ of dynamic logic.

Definition 6. \mathbb{T} *admits dynamic logic* if there exist, for each program sequence $\bar{y} \leftarrow \bar{q}$ and each formula $\varphi : D\Omega$, a formula $[\bar{y} \leftarrow \bar{q}] \varphi$ such that

$$[\bar{x} \leftarrow \bar{p}] (x_i \Rightarrow [\bar{y} \leftarrow \bar{q}] \varphi) \iff [\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] (x_i \Rightarrow \varphi)$$

for each program sequence $\bar{x} \leftarrow \bar{p}$ containing $x_i : \Omega$ and, dually, a formula $\langle \bar{y} \leftarrow \bar{q} \rangle \varphi$ such that

$$[\bar{x} \leftarrow \bar{p}] (\langle \bar{y} \leftarrow \bar{q} \rangle \varphi \Rightarrow x_i) \iff [\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow x_i).$$

(Since the internal logic is intuitionistic, one cannot simply define $\langle \bar{y} \leftarrow \bar{q} \rangle \varphi$ as $\neg[\bar{y} \leftarrow \bar{q}] \neg\varphi$.) The formulae $[\bar{y} \leftarrow \bar{q}] \varphi$ and $\langle \bar{y} \leftarrow \bar{q} \rangle \varphi$ are uniquely determined. A deduction system is obtained as a collection of lemmas in the internal logic. Most computational monads, with the exception of the continuation monad, admit dynamic logic [21].

Hoare triples for partial correctness $\{\varphi\} p \{\psi\}$ can be expressed in dynamic logic as $\varphi \Rightarrow [p] \psi$, and we now can also give a meaning to Hoare triples for total correctness by interpreting them as partial correctness plus termination:

$$[\varphi] p [\psi] := \varphi \Rightarrow (\langle p \rangle \top \wedge [p] \psi).$$

4 Exception Monads

We now proceed to develop a general treatment of monads that feature exceptions which can be raised and later caught. To begin, we give an equational definition in categorical terms, which is then translated into the do-notation. In the next section, this definition will be used to formulate Hoare-calculi for exception monads. Throughout, T will denote a strong monad.

There are two essential operations associated to exceptions: an operation *raise* that throws an exception and freezes the state, and a *catch* operation that returns a thrown exception, if any, and unfreezes the state, i.e. resumes normal execution of statements. Obvious variations such as an operation that catches only certain exceptions (e.g. exceptions only of a given class) and lets others pass are easily implemented by combinations of *catch* and *raise*. It will be convenient to give *raise* the polymorphic type $E \rightarrow TA$ (in the most basic example, *raise* is the right injection $E \rightarrow A + E$); of course, the result type A is in fact immaterial since *raise* does not return any results.

There is a certain amount of disagreement in the literature concerning whether raising exceptions should be regarded as a computational feature in its own right, or whether exceptions should just be recorded as part of the global

state. In the Java semantics developed in the LOOP project, the former approach is advocated [6, 9], while the latter option is preferred in [14, 26] (but not in [13], motivated by modularity considerations — compare this to Theorem 8 below). In terms of concrete monads, the LOOP project [9] uses the monad

$$J = \lambda A. (S \rightarrow (A \times S + E \times S + 1)),$$

with S the set of states and E the set of exceptions (the parameter A is thought of as the type of results), while

$$K = \lambda A. (S_E \rightarrow (A + 1) \times S_E + 1)$$

is implicitly used in [14, 26], where $S_E = S \times (E + 1)$, and where the use of $A + 1$ accommodates the fact that abnormally terminating statements do not return a value. The monad J is obtained by applying Moggi's exception monad transformer [11] to the usual state monad with non-termination, while K is the state monad with non-termination (and possibly undefined results) for the extended state set S_E . A simultaneous advantage and disadvantage of K is that *catch* is a monadic value, i.e. a statement, while in J , *catch* is a function on monadic values, i.e. a control structure. Thus, K contains monadic values that arbitrarily manipulate the state even though an exception has been thrown, or simultaneously throw an exception and return a value. Consequently, explicit Hoare rules are needed to force normal statements to be skipped in exceptional states [14, 26]; moreover, it becomes necessary to add normality of the state as a precondition to all of the usual Hoare rules. By contrast, in J the state is automatically frozen when an exception is thrown.

We shall model our treatment of generic exception monads along J rather than K , precisely because this allows better control over what monadic values do. Thus, we need to work with a *catch* operation of polymorphic type

$$TA \rightarrow T(A + E),$$

which takes a monadic value x and returns a monadic value that executes x but terminates normally (if at all), returning either a value returned by x or an exception raised by x .

We are now ready to present the announced categorical definition of exception monads. We begin with a short but somewhat mysterious definition and then proceed in opposite direction to the heuristics, explicating the definition stepwise into a number of intuitively convincing equations.

Definition 7. An *exception monad* is a strong monad T , together with a type E of *exceptions* and a natural transformation $catch : T \rightarrow T(- + E)$ such that

$$T \xrightarrow{catch} T(- + E) \xrightarrow[T \text{ inl}]{catch_{(-+E)}} T(- + E + E),$$

is an equalizer diagram of strong monad morphisms.

Recall that, given two strong monads S, T on a category \mathbf{C} , a (simplified) *strong monad morphism* is a natural transformation $\sigma : S \rightarrow T$, compatible with the remaining data in the sense that $\sigma\eta^S = \eta^T$, $\sigma\mu^S = \mu^T(\sigma * \sigma)$ (where $\sigma * \sigma = T\sigma\sigma_S = \sigma_T S\sigma$), and $\sigma_{(A \times _)} t_A^S = t_A^T(A \times \sigma)$ (e.g. [11]). Naturality of σ and compatibility with μ are, in terms of the associated Kleisli triples $(T, \eta^T, -^*)$ and $(S, \eta^S, -^+)$, equivalent to $\sigma f^* = (\sigma f)^+ \sigma$, i.e. σ is *compatible with binding*.

Next, let us point out which parts of Definition 7 are self-understood. Given a strong monad T , the functor $T(- + E)$ is made into a strong monad by taking $\eta^T \text{inl} : A \rightarrow T(A + E)$ as the unit, and a binding operator that transforms $f : A \rightarrow T(B + E)$ into $[f, \eta \text{inr}]^* : T(A + E) \rightarrow T(B + E)$; this is Moggi's monad transformer for exceptions [11] as implemented in the Haskell libraries [15] (of course, this presupposes that $A + E$ exists for all A). It is easy to check that $T \text{inl} : T \rightarrow T(- + E)$ is a strong monad morphism.

Thus, the actual conditions imposed by the definition are in particular

- $\text{catch} : T \rightarrow T(- + E)$ is a strong monad morphism. Compatibility with binding amounts to the equation

$$\text{catch } f^* = [\text{catch } f, \eta \text{inr}]^* \text{catch}.$$

We will see that this expresses the fact that in a sequential statement $p; q$, either p raises an exception, which is then passed on, and q is skipped, or p terminates normally and then q is executed. Compatibility of catch with the unit states that pure values do not throw exceptions: $\text{catch } \eta = \eta \text{inl}$.

- $\text{catch}_{(- + E)} \text{catch} = (T \text{inl}) \text{catch}$: this equation states that catch does not itself raise exceptions.

Finally, the fact that catch not only equalizes $\text{catch}_{(- + E)}$ and $T \text{inl}$, but is indeed their equalizer, can be captured equationally by means of the *raise* operation, which is conspicuously absent from the basic definition. Indeed we can *construct* this operation: we have a morphism $\eta : - + E \rightarrow T(- + E)$, which equalizes $\text{catch}_{(- + E)}$ and $T \text{inl}$ because catch is a monad morphism. Thus, we obtain a factorization $\text{catch } f = \eta$, where f is necessarily of the form $[\eta, \cdot]$; the *raise* operator is defined as the second component of this morphism:

$$\begin{array}{ccc} T & \xrightarrow{\text{catch}} & T(- + E) \\ \uparrow [\eta, \text{raise}] & \nearrow \eta & \\ - + E & & \end{array}$$

- i.e. the defining property of *raise* is the equation

$$\text{catch } \text{raise} = \eta \text{inr} \tag{1}$$

stating that raised exceptions are actually caught. In combination with binding compatibility of catch , this implies

$$\text{catch } [\eta, \text{raise}]^* = T(- + \nabla) \text{catch}_{(- + E)}, \tag{2}$$

where ∇ denotes the codiagonal $E+E \rightarrow E$; note that $T(-+\nabla) : T(-+E+E) \rightarrow T(-+E)$ is a strong monad morphism. From this equation, in turn, we can derive, using the fact that *catch* is monic, that

$$[\eta, \text{raise}]^* \text{catch} = \text{id} \quad (3)$$

— i.e. catching an exception can be undone by re-raising it.

Equations 2 and 3, together with the fact that *catch* equalizes $\text{catch}_{(-+E)}$ and $T \text{inl}$ and the obvious equation $T(-+\nabla)T \text{inl} = \text{id}$, amount to stating that

$$T \begin{array}{c} \xrightarrow{\text{catch}} \\ \xleftarrow{[\eta, \text{raise}]^*} \end{array} T(-+E) \begin{array}{c} \xrightarrow{\text{catch}_{(-+E)}} \\ \xleftarrow{T \text{inl}} \end{array} T(-+\nabla) T(-+E+E),$$

is a split equalizer diagram [1] of strong monad morphisms. Thus, we can equivalently describe an exception monad by means of equations 1 and 3 (the latter implies that *catch* is monic), together with the fact that *catch* is a strong monad morphism (since *catch* is monomorphic, it follows easily that $[\eta, \text{raise}]^*$ is also a strong monad morphism) and equalizes $\text{catch}_{(-+E)}$ and $T \text{inl}$.

The arising purely equational presentation of exception monads can be translated into the do-notation as explained in Section 2; the corresponding HASCASL specification is shown in Figure 2. The two imported specifications are the specification MONAD of monads as described in Section 2 and a specification SUMTYPE which provides $+$ as an infix sum type operator, with left and right injections *inl* and *inr*. The axioms (catch-ret) and (catch-seq) state that *catch* is a strong monad morphism (where (catch-seq) covers compatibility with binding as well as with the strength). Axiom (catch-raise) is Equation 1 above, (catch-catch) states that *catch* equalizes $\text{catch}_{(-+E)}$ and $T \text{inl}$, and (catchN) is Equation 3.

In the notation given in Figure 2, it becomes even more evident that all these axioms are properties that one would intuitively expect of *raise* and *catch*. In fact, as indicated above, the given equations come heuristically *before* Definition 7. Other expected properties follow; e.g., (catchN), (catch-seq), and (catch-raise) imply that ‘nothing happens after an exception is raised’, i.e. that

$$(\text{do } x \leftarrow \text{raise } e; p) = \text{raise } e. \quad (4)$$

An obvious way to construct exception monads is to use Moggi’s exception monad transformer as described above, i.e. to take $T = R(-+E)$ for a strong monad R , with

$$\text{catch} : R(-+E) \rightarrow R((-+E)+E)$$

being $R \text{inl}$. Surprisingly, this construction indeed *classifies* exception monads:

Theorem 8. *Let \mathbf{C} be a category with equalizers. Then every exception monad on \mathbf{C} is of the form $R(-+E)$ for some strong monad R on \mathbf{C} .*

Proof. Let T be an exception monad, and let R be the equalizer of the morphisms

$$\text{catch}, T \text{inl} : T \rightarrow T(-+E).$$

```

spec EXCEPTION = MONAD and SUMTYPE then
types  $E: Type, Ex: Monad$ 
var  $a, b: Type$ 
ops  $raise: E \rightarrow Ex\ Unit;$ 
 $catch: Ex\ a \rightarrow Ex\ (a + E);$ 
internal{
forall  $e: E; p: Ex\ a; q: a \rightarrow Ex\ b;$ 

- $catch\ (do\ x \leftarrow p; q\ x) =$   

 $do\ y \leftarrow catch\ p; case\ y\ of\ inl\ a \rightarrow catch\ q\ a$   

 $\quad\quad\quad | inr\ e \rightarrow ret\ (inr\ e)$  %(catch-seq)%
- $catch\ (ret\ x) = ret\ (inl\ x)$  %(catch-ret)%
- $catch\ (raise\ e) = ret\ (inr\ e)$  %(catch-raise)%
- $catch\ (catch\ p) = do\ y \leftarrow catch\ p; ret\ (inl\ y)$  %(catch-catch)%
- $p = do\ y \leftarrow catch\ p; case\ y\ of\ inl\ x \rightarrow ret\ x$   

 $\quad\quad\quad | inr\ e \rightarrow raise\ e$  %(catchN)%


}

```

Fig. 2. Equational specification of exception monads

This equalizer is preserved by the exception monad transformer $\lambda M. M(- + E)$, so that $R(- + E) \cong T$.

Thus, our definition of exception monads can be regarded as a complete equational characterization of the exception monad transformer. It may appear at first sight that this result is illicitly built into the definition, since the codomain of *catch* is $T(- + E)$. However, this is not the case: the result type of *catch* describes how exceptions are *observed*, and in this sense constitutes rather a minimal expectation. By contrast, the theorem above concerns the entirely different question of how exceptions are *represented* in computations, and the answer is not at all self-understood. One of its implications is that exceptions are never inextricably entwined with other notions of computation — they can always be regarded as added to the remaining computational features in a modular way.

The classification theorem can also be used to facilitate reasoning about exception monads; e.g. one can prove:

Corollary 9. *If T is an exception monad and $p : TA$ is a dsef computation, then p terminates normally, i.e. $catch\ p = (T\ inl)\ p$.*

(However, discardable computations may raise exceptions!).

Notation. In order to have intermediate results of a program sequence $\bar{x} \leftarrow \bar{p}$ available after a *catch*, the latter must be packaged up and explicitly returned. For this procedure, an abbreviated notation comes in handy: $catch\ \bar{x} \leftarrow \bar{p}$ denotes $catch\ (do\ \bar{x} \leftarrow \bar{p}; ret\ \bar{x})$.

Remark 10. One subtlety that we have omitted from the development so far is the fact that monad-independent computational logic [21, 23] uses a strengthened version of monads in the sense that binding is required to extend also to

partial morphisms, i.e. one has f^* for f partial. The unit laws for partial binding state that $f^*\eta$ agrees with f on the domain of f and behaves like f under binding (i.e. $(f^*\eta)^* = f^*$).

The results obtained so far are adapted to this setting as follows. Slightly surprisingly, the correct definition of a morphism σ of monads with partial binding turns out to require, in the notation used above, that the equation $\sigma f^* = (\sigma f)^+ \sigma$ holds strongly, i.e. as an equation between partial morphisms. Then the arising abstract definition of partial exception monads unravels in the same way as above. In particular, the only equation in Figure 2 that actually involves partial binding, (catch-seq), becomes a strong equation (to be read ‘one side is defined iff the other is, and then both sides are equal’). Moreover, the exception monad transformer indeed transforms monads with partial binding into monads with partial binding, and the analogue of Theorem 8 holds in categories with a notion of partial morphism, i.e. in dominional categories [19] with equalizers.

5 A generic Hoare calculus with abrupt termination

We now proceed to extend the partial and total generic Hoare calculi for monadic programs introduced in Section 2 to take into account exceptional termination, thus generalizing similar calculi [5, 7, 8]. The reason that the basic version of the generic Hoare calculus is insufficient for purposes of exceptional termination is that, due to Equation 4 above, we have

$$\{\} p \{\perp\}$$

whenever p raises an exception (indeed, this holds also e.g. in the ‘normal’ part of the calculi of [5, 7]). Thus, no reasonable statements can be made about what happens when an exception is raised. We remedy this problem by introducing an additional postcondition for exceptional termination, called the *abnormal postcondition* in opposition to the usual postcondition now called the *normal postcondition*.

This might raise the suspicion that the Hoare calculi of [21, 23] are in fact ‘insufficiently generic’, and that in reality every new computational feature requires a whole new Hoare calculus. Besides the examples given in [21, 23], the following heuristic argument supports our claim that this is not the case: the problem ‘ $\{\} \text{raise } e \{\perp\}$ ’ quoted above is due to the fact that exceptions are constants in the ambient monad, so that exceptional computations of type Ω do not actually contain any truth values. This phenomenon is unique to constant operations, and the only computational interpretation of constants known so far is precisely as exceptions. Thus, it appears that the need for substantially (i.e. other than in terms of additional axioms and rules for monad-specific program constructs) extended Hoare calculi will be limited to the case of exceptions.

We begin by introducing a partial Hoare calculus for abnormal termination in an exception monad T . A corresponding total Hoare calculus, which like the total Hoare calculus for normal termination requires additional assumptions on the

monad, will be treated further below. We denote the combination of precondition and normal and abnormal postcondition in the form

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

where the normal postcondition ψ is a stateful *formula* that may contain the result variables \bar{x} , while the abnormal postcondition S is stateful *predicate* on E (i.e. a function $E \rightarrow T\Omega$) and cannot use \bar{x} . This restriction reflects the fact that exceptional computations do not have a result; instead, the abnormal postcondition is concerned with a hitherto anonymous exception. The interpretation of such an *extended Hoare assertion* is 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 $S e$. Formally, $\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi \parallel S\}$ abbreviates

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

The associated Hoare calculus subsumes the generic Hoare calculus [23], since one can show that

$$\{\phi\} \bar{x} \leftarrow \bar{p} \{\psi\} \iff \{\phi\} \bar{x} \leftarrow \bar{p} \{\psi \parallel \top\}.$$

Figure 3 shows a set of proof rules for extended Hoare assertions. Most of the rules except (catch) and (raise) have counterparts in the ‘normal’ generic Hoare calculus; note in particular the single composition rule (seq), which is similar to the one given in [8]. In the conjunction and weakening rules, notations such as $S_1 \wedge S_2$ stand for the pointwise operations, here: $\lambda e : E. (S_1 e) \wedge (S_2 e)$. The (Y) rule refers to the fixed-point operator Y (cf. Section 2); this rule applies only to cpo-monads. Application of the Y operator to F requires implicitly that F has the continuous function type $(A \xrightarrow{c} TB) \xrightarrow{c} (A \xrightarrow{c} TB)$. The square brackets indicate reasoning with local assumptions, discharged by application of the rule. Soundness of the rule is a consequence of the fact that satisfaction of extended Hoare assertions is an admissible predicate. Using this rule, one easily derives rules for particular recursive functions, e.g. the usual while rule or a corresponding rule for the iteration construct described in Section 2,

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

(A corresponding rule is listed as a basic rule in the generic Hoare calculus [23], which can indeed be improved by moving to a (Y)-like rule.) The exception-monad specific rules (catch) and (raise) state that *catch* catches a thrown exception but otherwise does not affect the enclosed program, and that *raise* really throws the given exception.

The calculus is sound w.r.t. the coding of Hoare triples as formulas in the internal language:

Theorem 11. *If an extended Hoare assertion is derivable in an exception monad (cpo-monad) by the rules of Figure 3 excluding (including) (Y), then the translated formula is derivable in the internal language.*

The proof uses the definition of extended Hoare assertions via standard Hoare triples, the generic Hoare rules [23], and the equations of Figure 2. Completeness of the calculus is the subject of further research; we conjecture that a completeness proof for the Hoare logic of [23] will lead to completeness of the extended calculus, since each of the equations in Figure 2 is reflected in one of the rules of Figure 3.

$$\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{(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'\}} \\
\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{(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\}} \quad \text{(Y)} \frac{[\{\phi\} x \leftarrow p \ y \ \{\psi \parallel S\}] \quad \vdots}{\{\phi\} x \leftarrow (F \ p) \ y \ \{\psi \parallel S\}} \frac{\{\phi\} x \leftarrow Y(F) \ y \ \{\psi \parallel S\}}{\{\phi\} x \leftarrow Y(F) \ y \ \{\psi \parallel S\}} \\
\text{(ctr)} \frac{\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\}} \quad \text{(dsef)} \frac{p \ \text{dsef}}{\{\phi\} x \leftarrow p \ \{\phi \wedge x = p \parallel \perp\}} \\
\text{(raise)} \frac{}{\{\} \text{raise } e_0 \ \{\perp \parallel \lambda e. e = e_0\}} \quad \text{(stateless)} \frac{}{\{\text{ret } \phi\} q \ \{\text{ret } \phi \parallel \lambda e. \text{ret } \phi\}} \\
\text{(catch)} \frac{\{\phi\} \bar{x} \leftarrow \bar{p} \ \{\psi \parallel S\}}{\{\phi\} y \leftarrow (\text{catch } \bar{x} \leftarrow p) \ \{\text{case } y \ \text{of } \text{inl } \bar{x} \rightarrow \psi \mid \text{inr } e \rightarrow S \ e \parallel \perp\}}
\end{array}$$

Fig. 3. The generic Hoare calculus for partial exception correctness

Total extended Hoare assertions for reasoning about total correctness are, as in the basic case, encoded in generic dynamic logic [21]; this requires additionally that the monad admits dynamic logic. In this respect, exceptions do not cause additional problems:

Theorem 12. *If a strong monad T admits dynamic logic, then so does $T(-+E)$.*

If an exception monad T admits dynamic logic, then we can define combined total correctness assertions for normal and abnormal termination by

$$[\phi] \bar{x} \leftarrow \bar{p} [\psi \parallel S] \equiv \{\phi\} \bar{x} \leftarrow \bar{p} \{\psi \parallel S\} \wedge \langle \text{catch } p \rangle \top.$$

A Hoare calculus for such *extended total Hoare assertions* can be derived from the equational axioms for exception monads by means of the proof system for generic dynamic logic [21]. As in the basic case, the rules are essentially the same as for extended partial Hoare assertions, with the exception of the (Y) rule (and, of course, the rules for constructs such as *while* and *iter* derived from it). In fact, there does not seem to be an immediately obvious total analogue of (Y); however, one can easily prove e.g. a total Hoare rule for *iter* (which then specializes to a corresponding total *while* rule (while-total) by taking $\text{while } b \ p = \text{iter } b \ p$):

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

In JML, the effect of a statement is specified by giving clauses for the precondition (**assumes**), the normal postcondition (**ensures**), the abnormal postcondition (**signals**), and a precondition for non-termination (**diverges**) which must hold before execution of a statement in cases where the statement hangs [8]. The specification $\{\mathbf{assumes} = \phi, \mathbf{ensures} = \psi, \mathbf{signals} = S, \mathbf{diverges} = \delta\}$ for a statement p can be expressed in the notation above as

$$\{\phi\} p \ \{\psi \parallel S\} \quad \text{and} \quad [\phi \wedge \neg\delta] p \ [\top \parallel \lambda e. \top]$$

(The observation that the **derives** clause can be coded out in this way is made already in [8], and indeed the calculus discussed there forces this coding to be used in proofs. Of course, taken literally, this works only classically, but intuitionistically, one would at any rate prefer a condition that guarantees termination, replacing $\neg\delta$, over one that is entailed by non-termination.) By consequence, also the Hoare calculus of [7] is expressible in our calculus (ignoring, for the time being, class constraints on the involved exceptions; given a formal description of the class mechanism as described e.g. in [6], such constraints can be expressed in the postcondition as carried out in [5]). Explicitly, we can put, e.g.,

$$\begin{aligned} \{\phi\} p \ \{\mathbf{exception}(S)\} &\equiv \{\phi\} p \ \{\top \parallel S\} \quad \text{and} \\ [\phi] p \ [\mathbf{exception}(S)] &\equiv [\phi] p \ [\perp \parallel S], \end{aligned}$$

thus obtaining conditions stating that, under precondition ϕ ,

- if p terminates abnormally with exception e , then the resulting state satisfies $S e$ (*partial exception correctness*), and
- p terminates abnormally with exception e in a state that satisfies $S e$ (*total exception correctness*), respectively.

In fact, $\{\phi\} x \leftarrow p \{\psi \parallel S\}$ is equivalent to the conjunction of $\{\phi\} x \leftarrow p \{\psi\}$ and $\{\phi\} p \{\text{exception}(S)\}$ (while no such simplification holds in general for total extended Hoare assertions).

Example 13. The exceptional Hoare triple $\{\phi\} p \{\text{exception}(S)\}$ is made explicit in $T(- + E)$ for concrete T as follows:

- T the state monad: if execution of p from a state s satisfying ϕ terminates with exception e , then $S e$ holds in the poststate.
- T the non-deterministic state monad: whenever p possibly terminates exceptionally in state s' with exception e , then $S e$ holds in s' .
- T the input monad: whenever p throws an exception e after reading a string of inputs, e satisfies $S e$.

While there is, due to the fact that the calculus of [5, 7] is tailored specifically to Java, no precise one-to-one correspondence between the rules listed there and our generic rules, the central features of the former do in fact drop out of the generic calculus. In particular, the composition rules of [7] can be derived from rule (seq) in Figure 3 and its total analogue, and the *partial exception while rule* [5] is the projection of the generic while rule (obtained as a specialization of rule (iter) above) to partial exception correctness (i.e. normal correctness is just dropped in the conclusion). More interesting is the derivation of the *total exception while rule*: this rule (reformulated in analogy to the *total break while rule* [7]) translates into our calculus as

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

(with exceptional Hoare triples already coded out). The premises can be gathered into the single total extended Hoare assertion

$$[\phi \wedge b \wedge t = n] p [\phi \wedge b \wedge t < n \parallel S],$$

and from this we can indeed draw the required conclusion by means of the generic total while rule (while-total) (see above), noting that in the normal postcondition, we get the contradiction $b \wedge \neg b$. Similarly, the Hoare rules for JML method specifications [8] are in direct correspondence with the generic rules of our calculus. All this is by no means intended to disparage JML (which constitutes a much wider effort involving many aspects of a concrete programming language), but rather goes to show that the kernel of known specialized Hoare calculi with exceptions is covered by our generic mechanism.

6 Conclusion and Future Work

We have generalized existing Hoare calculi for the Java monad [5, 7, 8] to a monad-independent Hoare calculus for exception monads, expressed within the specification language HASCASL. To this end, we have extended previous work [21, 23] on monad-independent calculi by adding postconditions for abrupt termination; this was based on an equational characterization of Moggi’s exception monad transformer which arose from a rather striking categorical formulation.

This extension of the monad-independent Hoare logic is necessary for two reasons:

- the *catch* operation is not algebraic, but rather acts on top of the monad.
- exceptions are constants, so that the basic monad independent calculus cannot make statements about exceptional post-states

We argue that both these reasons, in particular the latter, apply uniquely to exceptions as a computational feature. By contrast, other computational effects are purely algebraic [18], and moreover based on non-constant algebraic operations. Hence, we expect that the genericity of our approach will allow for an easy integration of further monadic computational effects without any substantial extensions of the overall formalism as in the case of exceptions. Such additional computational effects include in particular input/output [17] and concurrency [3, 16]. While the corresponding monads have been designed for use with the functional language Haskell, we do not anticipate major obstacles in their re-use as extensions of the Java monad [9].

Acknowledgements

This work forms part of the DFG-funded project HasCASL (KR 1191/7-1). The authors wish to thank Christoph Lüth for useful comments and discussions.

References

- [1] M. Barr and C. Wells, *Toposes, triples and theories*, Springer, 1984.
- [2] M. Bidoit and P. D. Mosses, *CASL user manual*, LNCS, IFIP Series, vol. 2900, Springer, 2003.
- [3] K. Claessen, *A poor man’s concurrency monad*, J. Funct. Programming **9** (1999), 313–323.
- [4] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), no. 10, 576–580.
- [5] M. Huisman, *Java program verification in higher order logic with PVS and Isabelle*, Ph.D. thesis, University of Nijmegen, 2001.
- [6] M. Huisman and B. Jacobs, *Inheritance in higher order logic: Modeling and reasoning*, Theorem Proving in Higher Order Logics, LNCS, vol. 1869, Springer, 2000, pp. 301–319.

- [7] ———, *Java program verification via a Hoare logic with abrupt termination*, Fundamental Approaches to Software Engineering, LNCS, vol. 1783, Springer, 2000, pp. 284–303.
- [8] 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.
- [9] ———, *Coalgebras and Monads in the Semantics of Java*, Theoret. Comput. Sci. **291** (2003), 329–349.
- [10] E. Moggi, *Categories of partial morphisms and the λ_p -calculus*, Category Theory and Computer Programming, LNCS, vol. 240, Springer, 1986, pp. 242–251.
- [11] ———, *An abstract view of programming languages*, Tech. Report ECS-LFCS-90-113, Univ. of Edinburgh, 1990.
- [12] ———, *Notions of computation and monads*, Inform. and Comput. **93** (1991), 55–92.
- [13] T. Nipkow, *Jinja: Towards a comprehensive formal semantics for a Java-like language*, Proc. Marktobderdorf Summer School 2003, IOS Press, to appear.
- [14] ———, *Hoare logics in Isabelle/HOL*, Proof and System-Reliability, Kluwer, 2002, pp. 341–367.
- [15] S. Peyton-Jones (ed.), *Haskell 98 language and libraries — the revised report*, Cambridge, 2003, also: J. Funct. Programming **13** (2003).
- [16] S. Peyton Jones, A. Gordon, and S. Finne, *Concurrent Haskell*, Principles of Programming Languages, ACM Press, 1996, pp. 295–308.
- [17] S. Peyton Jones and P. Wadler, *Imperative functional programming*, Principles of Programming Languages, ACM Press, 1993, pp. 71–84.
- [18] G. Plotkin and J. Power, *Notions of computation determine monads*, Foundations of Software Science and Computation Structures, LNCS, vol. 2303, Springer, 2002, pp. 342–356.
- [19] L. Schröder, *Classifying categories for partial equational logic*, Category Theory and Computer Science, ENTCS, vol. 69, 2002.
- [20] ———, *Henkin models of the partial λ -calculus*, Computer Science Logic, LNCS, vol. 2803, Springer, 2003, pp. 498–512.
- [21] L. Schröder and T. Mossakowski, *Monad-independent dynamic logic in HASCASL*, J. Logic Comput., to appear. Preliminary version in Recent Developments in Algebraic Development Techniques, LNCS 2755 (2003), Springer, pp. 425–441.
- [22] 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.
- [23] ———, *Monad-independent Hoare logic in HASCASL*, Fundamental Aspects of Software Engineering, LNCS, vol. 2621, 2003, pp. 261–277.
- [24] 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
- [25] H. Thielecke, *Categorical structure of continuation passing style*, Ph.D. thesis, University of Edinburgh, 1997.
- [26] D. von Oheimb, *Hoare logic for Java in Isabelle/HOL*, Concurrency and Computation: Practice and Experience **13** (2001), 1173–1214.