

Monad-based logics for computational effects

Till Mossakowski, DFKI Lab Bremen, Germany

joint work with

Lutz Schröder, Sergey Goncharov, Denis Walter

AMAST 2006, 08 July 2006

Dedicated to Joseph Goguen (28 June 1941 - 03 July 2006)



Overview

- Monads: definition and examples
- The basic term language, do-Notation
- Global dynamic judgements
- **Hoare calculus**, side-effect freeness (works for all monads)
- **Dynamic logic** (more expressive, works for some monads)
- The exception monad and abnormal termination
- The resumption monad and parallel composition
- Conclusion and outlook

Introduction

- computational effects complicate the verification of programs
- normally, each effect (combination) requires a new logic
- Monads:
 - algebraic encapsulation of **generic side effects** (Moggi 1991)
 - used for '**imperative functional programming**' in Haskell (Wadler 97)
- Specification and verification of imperative programs, e.g. what means

$$\{x = 4\} x := x + 1 \{x = 5\}$$

in a monad-based setting?

The Origins: Monads Generalize Algebra

Given an equational theory (S, OP, E) , define

- $TX = T_{(S,OP)}(X)/\equiv_E$ (quotient term algebra)
- $\eta_X : X \rightarrow TX$, $\eta(x) = [x]$ (insertion of variables)
- a substitution $h : X \rightarrow TY$ can be extended to all of TX

$$h^* : TX \rightarrow TY \text{ (term evaluation)}$$

- **Kleisli composition** of substitutions: $f : X \rightarrow TY$ and $g : Y \rightarrow TZ$ can be composed as

$$X \xrightarrow{f} TY \xrightarrow{g^*} TZ$$

Monads: Formal Definition

A **Kleisli triple** $\mathbb{T} = (T, \eta, _*)$ on a category \mathbf{C} is given by

- $T : \text{Ob } \mathbf{C} \rightarrow \text{Ob } \mathbf{C}$
- **unit morphisms** $\eta_A : A \rightarrow TA$, and
- $_*$ with $f^* : TA \rightarrow TB$ for $f : A \rightarrow TB$, such that

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

In Haskell, given $p : TA$ and $q : A \rightarrow TB$, $p \gg q$ is $q^*(p)$

A **strength** is a (coherent) natural transformation

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

Computational Monads

Following (Moggi 1991),

- Notions of **computation** are modeled by **strong monads**
- Functions with side-effects are 'Kleisli morphisms'

$$A \rightarrow TB$$

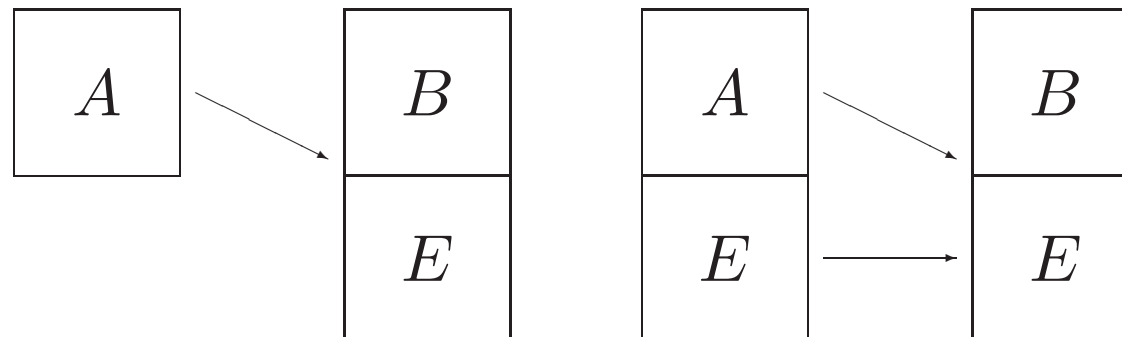
Example: Exceptions

$$TA = A + E$$

$$\eta_A = \text{inl} : A \hookrightarrow A + E$$

Given $f : A \rightarrow B + E$, how to construct $f^* : A + E \rightarrow B + E$?

$$\begin{aligned} f^*(\text{inl}(a)) &= f(a) \\ f^*(\text{inr}(e)) &= e \end{aligned}$$



Example: States

$$T A := S \rightarrow (A \times S)$$

$$\eta(x) = \lambda s : S \bullet (x, s)$$

$$f : A \rightarrow TB = A \rightarrow (S \rightarrow (B \times S))$$

$$f^* : TA \rightarrow TB$$

$$f^*(y : TA) = S \xrightarrow{y} A \times S \xrightarrow{\text{uncurry}(f)} B \times S$$

Monads: Further Examples

- the **Java monad**: $TA = (S \rightarrow (A \times S + E \times S + 1))$
(E is parametrized over return values) (Jacobs, Poll 2003)
- (finite) **non-determinism**: $TA = \mathcal{P}_{fin}(A)$
- **non-deterministic stateful computations**: $TA = (S \rightarrow \mathcal{P}_{fin}(A \times S))$
- interactive **input**: $TA = \mu\gamma. A + (U \rightarrow \gamma)$
- **output, queues**: $TA = U^* \times A$
- **continuations (“goto”)**: $TA = (A \rightarrow R) \rightarrow R$ (Moggi 1990)
- **resumptions (traces/concurrency)** over a monad M :
 $RA = \mu\gamma. M(A + \gamma)$
(Moggi 1990, Papaspyrou 1995, Filinski 1999, Papaspyrou 2001)
- **backtracking** over a monad T (Kiselyov et al. 2005):
 $Back A = \Pi B : Type. (A \rightarrow TB \rightarrow TB) \rightarrow TB \rightarrow TB$

The basic term language; do-Notation

do-Notation for Kleisli Composition (Moggi 1991)

Given $p : TA$, $q : A \rightarrow TB$,

$\text{do } x \leftarrow p; q(x)$ abbreviates $q^*(p) : TB$

Haskell-like example, using $S = \Pi A. \text{Loc } A \rightarrow A$:

```
inc l = do x <- !l
        l := x+1
```

```
!_ : Loc A -> TA
!l (s) = (s(l), s)
```

```
main = do l <- new 4
          inc l
          x <- !l
          putStrLn (show x)
```

```
_ := _ : Loc A -> A -> T1
l := a (s) = ((), s[l |-> a])
```

Assumptions for Monad-based Logic

We work with a **strong monad** over a **cartesian category** that

- has a **Heyting algebra** object Ω (for the basic logic)
- additionally may be cartesian closed (for interpreting higher-order functions)
- or may be a partial cartesian closed category or a topos (for interpreting higher-order logic, like in `HASCASL`).

The monad needs to be a CPO-monad if we need interpret recursion. Additional infrastructure (like universal object, algebraic compactness) needed to interpret recursive types. Prototypical example: **pCPO**.

The Basic Language: Types and Terms

$$A ::= 1 \mid \Omega \mid TA \mid A \times A \mid S$$

$$\begin{array}{l}
 \text{(var)} \frac{}{\Gamma, x : A \triangleright x : A} \quad \text{(app)} \frac{f : A \rightarrow B \in \Sigma \quad \Gamma \triangleright t : A}{\Gamma \triangleright f(t) : B} \quad \text{(1)} \frac{}{\Gamma \triangleright () : 1} \\
 \text{(pair)} \frac{\Gamma \triangleright t : A \quad \Gamma \triangleright u : B}{\Gamma \triangleright (t, u) : A \times B} \quad \text{(fst)} \frac{\Gamma \triangleright t : A \times B}{\Gamma \triangleright fst(t) : A} \quad \text{(snd)} \frac{\Gamma \triangleright t : A \times B}{\Gamma \triangleright snd(t) : A} \\
 \text{(do)} \frac{\Gamma \triangleright p : TA \quad \Gamma, x : A \triangleright q : TB}{\Gamma \triangleright \text{do } x \leftarrow p; q : TB} \quad \text{(ret)} \frac{\Gamma \triangleright t : A}{\Gamma \triangleright \text{ret } t : TA} \\
 \text{(\top)} \frac{}{\Gamma \triangleright \top : \Omega} \quad \text{(\perp)} \frac{}{\Gamma \triangleright \perp : \Omega} \quad \text{(\neg)} \frac{\Gamma \triangleright \varphi : \Omega}{\Gamma \triangleright \neg\varphi : \Omega} \quad \text{etc.}
 \end{array}$$

$$[\text{do } x \leftarrow p; q] = [\Gamma] \xrightarrow{\langle id, [p] \rangle} [\Gamma] \times TA \xrightarrow{t_{[\Gamma], A}} T([\Gamma] \times A) \xrightarrow{[q]} TB$$

Global dynamic judgements

Global dynamic judgements

do $\bar{x} \leftarrow \bar{p}$; q abbreviates do $x_1 \leftarrow p_1$; ... do $x_n \leftarrow p_n$; q

A statement $[[\bar{x} \leftarrow \bar{p}]]_G \phi$, i.e.

‘ ϕ holds **globally** after executing \bar{p} and binding the result to \bar{x} ’

abbreviates

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

(Schröder, Mossakowski 2003)

Examples

$\llbracket x \leftarrow p \rrbracket_G \phi$ means. . .

- **State:** ϕ holds for the result of p after execution from any initial state
- **Non-determinism:** ϕ holds for all possible results of p
- **Exceptions:** if p terminates normally, then ϕ holds for its result
- **Input:** ϕ holds for the result of p after any combination of inputs
- **Non-deterministic state:** ϕ holds for all possible results of p after execution from any initial state
- **Resumptions:** if p terminates, ϕ holds in the base monad after executing all steps of p

**Hoare calculus;
side-effect freeness**

Hoare calculus

- Hoare triple

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

abbreviates

$$\llbracket a \leftarrow \phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \psi \rrbracket_G a \Rightarrow b.$$

- pre/post-conditions:

- **stateful formulas**: $\phi, \psi : T\Omega$
- **deterministically side effect free** (subtype $D\Omega$)

- Results \bar{x} can be used in post-condition

(Harrison 2001, Schröder, Mossakowski 2003)

Hoare Triples: Examples

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-determinism monad** iff, whenever ϕ holds possibly, then ψ holds for all possible results x of p ;
- in the exception monad iff, whenever ϕ holds and p terminates normally, returning x , then ψ holds for x ;
- in the **interactive input monad** iff, whenever ϕ holds and p returns x after reading some sequence of inputs, then ψ holds for x .
- in the **non-deterministic state monad** iff, whenever ϕ holds possibly in a state s , then ϕ holds after execution of p for all possible results x .

Example

$$\{\text{do } x \leftarrow !l; \text{ret } x = 4\} \text{inc } l \{\text{do } x \leftarrow !l; \text{ret } x = 5\}$$

or shortly, since $!l$ is side-effect free,

$$\{!l = 4\} \text{inc } l \{!l = 5\}$$

Side-effect freeness

- p **discardable** (Thielecke 1997) ('not modifying the state') if

$$(\text{do } p; \text{ret}()) = \text{ret}().$$

- p is **copyable** (Thielecke 1997) ('deterministic') if

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

- p is **deterministically side effect free** (dsef) if it is discardable, copyable and commutes with all such programs (Schröder, Mossakowski 2004)
- Related: p exists (Moggi 1991), p innocent (Harrison 2001), purity of methods in JML (Leavens et al. 1999)

Discardability: examples

ϕ discardable means

- **State monad:** ϕ reads the state, but does not change it
- **Exceptions:** ϕ terminates normally
- **Non-determinism:** ϕ does not fail
(whereas ϕ dsef means that ϕ yields **exactly** one result)
- **In-/Output:** ϕ does not read/write
- **Resumptions:** ϕ does one step, which is discardable in the base monad
- **Continuations:** ϕ is stateless (i.e. $ret \top$ or $ret \perp$) — not useful

What is missing?

We would like to speak about

- termination
- total correctness
- weakest preconditions
- reasoning about local “state”

All this can be done with **dynamic logic**.

Dynamic logic

Dynamic Logic

- Modal logic with operators

$$[\bar{x} \leftarrow \bar{p}] \quad \text{and} \quad \langle \bar{x} \leftarrow \bar{p} \rangle$$

read ‘holds always/possibly after execution of \bar{p} ’.

- again: results \bar{x} can be used later
- also called **evaluation logic**:
 - Pitts (1991): local semantics using monads **and hyperdoctrines**
 - Moggi (1991): **global** semantics using only monads

here: **local axiomatic** semantics using only monads. (Stronger than Hoare logic, but needs extra assumptions — which usually hold.)

What Dynamic Logic Can Express

- p terminates: $\langle p \rangle \top$
- total correctness $[\varphi] \bar{x} \leftarrow \bar{p} [\psi]$ is expressed as

$$\varphi \Rightarrow ((\langle \bar{p} \rangle \top) \wedge [\bar{x} \leftarrow \bar{p}] \psi)$$

- $[\bar{x} \leftarrow \bar{p}] \varphi$
is the **weakest precondition** for ensuring φ after execution of \bar{p}
- reasoning about local “state”
 $\neg \text{empty} \wedge [\text{enq } z; x \leftarrow \text{deq}] \varphi \Leftrightarrow \neg \text{empty} \wedge [x \leftarrow \text{deq}; \text{enq } z] \varphi$

Dynamic Logic: Axiomatic Semantics

- Again: formulas are deterministically side-effect free
- T admits PDL, if $[\bar{y} \leftarrow \bar{q}] \phi$ determined by

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

exists (Schröder, Mossakowski 2004)

- Under suitable conditions concerning extraction of **abstract** states, existence of $[\bar{y} \leftarrow \bar{q}] \phi$ can be proved.
- In the classical case, $\langle \bar{x} \leftarrow \bar{p} \rangle \phi$ is $\neg[\bar{x} \leftarrow \bar{p}] \neg\phi$
- Axioms and rules of dynamic logic follow

Dynamic Logic: Examples

$[x \leftarrow p] \phi$ means

- in the **state monad**: after executing p from the **given state**, ϕ holds for x
- in the **non-deterministic state monad**: after all possible executions of p from the given state returning value x , ϕ holds for x
($\langle x \leftarrow p \rangle \phi$: . . . **some** execution of p . . .)
- in the **interactive input monad**: for any sequence of inputs, if execution of p returns, ϕ holds for x
- the **continuation monad** does not admit dynamic logic

Dynamic Logic: Proof Rules

| | | |
|------------------|--|--|
| | $\text{(nec)} \quad \frac{\varphi}{[\bar{x} \leftarrow \bar{p}] \varphi} \quad \bar{x} \text{ not free in assumptions}$ | $\text{(mp)} \quad \frac{\varphi \Rightarrow \psi; \quad \varphi}{\psi}$ |
| (K) | $[\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi) \Rightarrow [\bar{x} \leftarrow \bar{p}] \varphi \Rightarrow [\bar{x} \leftarrow \bar{p}] \psi$ | |
| (seq \square) | $[\bar{x} \leftarrow \bar{p}; y \leftarrow q] \varphi \iff [\bar{x} \leftarrow \bar{p}] [y \leftarrow q] \varphi$ | |
| (ctr \square) | $[x \leftarrow p; y \leftarrow q] \varphi \iff [y \leftarrow (\text{do } x \leftarrow p; q)] \varphi$ | $(x \notin FV(\varphi))$ |
| (ret \square) | $[x \leftarrow \text{ret } t] \varphi \iff \varphi[t/x]$ | |
| (dis) | $[x \leftarrow p] \psi \iff \psi$ | $p \text{ dsef}, x \text{ not free in } \psi$ |
| (copy) | $[x \leftarrow p; y \leftarrow p] \psi \iff [x \leftarrow p] \psi[x/y]$ | $p \text{ dsef}$ |
| (cong-ret) | $\text{ret}(t \iff u) \Rightarrow (\varphi[t/x] \iff \varphi[u/x])$ | |
| (\square) | $[a \leftarrow ([b \leftarrow p] b)] \text{ret}(t \Rightarrow a) \iff [a \leftarrow p] \text{ret}(t \Rightarrow a)$ | $p : T\Omega$ |
| (unit) | $[x \leftarrow \varphi] \text{ret } x \iff \varphi$ | |
| (CC) | $\varphi[t/x] \iff \varphi[u/x]$ | for $CC \vdash t = u$ |
| (taut) | $\text{ret } t$ | $t : \Omega$ a tautology |

$$CC = \{\text{fst}(x, y) = x; \text{snd}(x, y) = y; (\text{fst}(x), \text{snd}(x)) = x; x : 1 = ()\}$$

Dynamic Logic: Completeness

Theorem. Let T be a **simple** strong monad over a cartesian category with a Boolean algebra object admitting PDL.

If all basic operations have T -free argument types, the calculus for dynamic logic is complete

Proof. Via a term model construction. (Mossakowski, Schröder, Goncharov 2006)

Simple Monads

A monad is **simple**, if the formula defining $\llbracket \bar{x} \leftarrow \bar{p} \rrbracket_G \phi$

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

is equivalent to

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

Crucial consequence: $[x \leftarrow p; y \leftarrow q] \varphi \iff [y \leftarrow (\text{do } x \leftarrow p; q)] \varphi$

Prop. A monad axiomatized equationally is simple if each equation contains the same variables on both sides.

Simple Monads Exception, state, nondeterminism, Java monad

Non-simple Monads Continuation, concurrency, backtracking, Abelian groups

Termination and Abstract States

A program p **terminates** if

$$\llbracket \bar{x} \leftarrow \bar{q}; p \rrbracket_G \phi \quad \text{implies} \quad \llbracket \bar{x} \leftarrow \bar{q} \rrbracket_G \phi$$

for each program sequence $\bar{x} \leftarrow \bar{q}$ and each $\phi : \Omega$.

A **state** is a program $s : T1$ such that s terminates and such that, for each dsef program $p : DA$, there exists $a : A$ such that

$$(\text{do } s; p) = \text{do } s; \text{ret } a.$$

State Discloser

A state s is called **forcible** if

$$s = \text{do } p; s$$

for each terminating program p .

A dsef program $d : DS$ is called a **state discloser** if the term

$$\text{do } x \leftarrow d; x$$

of type $T1$ is discardable (S is the type of forcible states).

A Characterization Theorem

Theorem. If \mathbb{T} has a state discloser, then $DA \cong (S \rightarrow A)$.

Moreover, under further conditions, \mathbb{T} admits dynamic logic.
(Schröder, Mossakowski 2004)

Proof. The isomorphism κ maps $y : DA$ to

$$\kappa(y) := \lambda s : S \bullet \iota a : A. ((\text{do } s; y) = (\text{do } s; \text{ret } a))$$

and its inverse maps $f : S \rightarrow A$ to

$$\kappa^{-1}(f) := \text{do } x \leftarrow d; \text{ret}(f x).$$

A Hoare calculus for total correctness

- Dynamic Logic can express termination of p as

$$\langle p \rangle \top$$

- Code 'total Hoare triples' $[\varphi] p [\psi]$ as

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

- Prove rules such as

$$\begin{array}{c}
 t : DA \\
 \text{---} < \text{---} : A \times A \rightarrow \Omega \text{ is well-founded} \\
 [\phi \wedge b \wedge (t = z)] p [\phi \wedge (t < z)] \\
 \hline
 \text{(while)} \quad [\phi] \text{ while } b p [\phi \wedge \neg b]
 \end{array}$$

Example: The reference monad

spec REFERENCE = CPOMONAD **then**

var $a : Cpo$

types $R : CpoMonad; Loc\ a : Flatcpo$

ops $!_{--} : Loc\ a \xrightarrow{c} ? R\ a;$

$-- := -- : Loc\ a \xrightarrow{c} a \xrightarrow{c} R\ 1$

forall $x, y : a; r, s : Loc\ a$

- $dsef(!r)$
- $[]\ r := x\ [x = !r]$
- $[\neg r = s \wedge x = !r]\ s := y\ [x = !r]$

The dynamic reference monad

spec DYNAMICREFERENCE = REFERENCE **then**

var $a, b : \text{Type}$

op $\text{new} : a \xrightarrow{c} R(\text{Loc } a)$

forall $x, y : a; r : \text{Loc } a; p : R b$

- $[] r \leftarrow \text{new } x [x = !r]$
- $[x = !r] s \leftarrow \text{new } y [\neg r = s \Rightarrow x = !r]$
- $[] r \leftarrow \text{new } x; p; s \leftarrow \text{new } y [\neg r = s]$

Monad combination ‘Napoleon’

spec NONDETERMINISM = CPOMONAD **then**

var $a : Cpo$

ops $fail, chaos : N a;$

$--- \parallel ---, ---sync--- : N a \xrightarrow{c} N a \xrightarrow{c} N a$

forall . . .

spec NONDETERMINISTICDYNAMICREFERENCE =

DYNAMICREFERENCE **with** $R \mapsto NR$

and NONDETERMINISM **with** $N \mapsto NR$

Meta-Example: Euclid's algorithm

Can now prove total correctness of Dijkstra's non-deterministic implementation

```
r ← new x;  
s ← new y;  
while ret ( $\neg !r = !s$ )  
  (if ret ( $!r > !s$ ) then  $r := !r - !s$  else fail)  
   $\square$   
  (if ret ( $!s > !r$ ) then  $s := !s - !r$  else fail)  
ret ( $!r$ )
```

(in a **loose** non-deterministic reference monad!)

The exception monad and abnormal termination

Distinguishing exceptions from non-termination

- Basic operations of exception monads:

$$\textit{raise} : E \rightarrow TA \quad \text{and} \quad \textit{catch} : TA \rightarrow T(A + E)$$

- Problem: No distinction between exceptions and non-termination for calculi presented so far, i.e.

$$\{\} \textit{raise} e \{\perp\}$$

- Idea: Equational characterization of exception monads + introduction of ‘abnormal’ postconditions (Schröder, Mossakowski 2004a)

Extended Hoare Triples

Now statements about exceptional results possible:

- Introduce extended Hoare triples

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

where $S : E \rightarrow T\Omega$ is a predicate on exception values and must not mention x .

- Interpret them as

$$\{\phi\} y \leftarrow \text{catch } p \left\{ \begin{array}{l} \text{case } y \text{ of } \\ \quad \text{inl } x \rightarrow \psi \\ \quad \text{inr } e \rightarrow S e \end{array} \right\}$$

Verification of a Pattern Match Algorithm (Walter 2005)

The following Java program looks for a pattern in a base string:

```
class Pattern {
  int [] base;
  int [] pattern;
  int find_pos () {
    int p = 0; s = 0;
    while (true)
      if (p == pattern.length) return s;
      else if (s + p == base.length) raise PatternNotFound;
      else if (base!!(s + p) == pattern!!p) p++;
      else { s++; p = 0; }
  }
}
```

Translation to Monadic Style

```
pmatch base pattern = mbody ( do {  
  p ← new 0;      s ← new 0;  
  while (ret ⊤)  
    if !p == length pattern  
    then raise (MRet !s)  
    else if !s + !p == length base  
    then raise PatternNotFound  
    else if base!!(!s+!p) == pattern!!(!p)  
    then p := (!p+1)  
    else do { s :=(!s+1); p := 0 } )
```

Proof Obligation

$$\begin{aligned} \square p \ [\perp \ \parallel \ \lambda e. \text{case } e \text{ of} \\ \quad MRet\ i \rightarrow MPOS\ i \wedge \forall j. MPOS\ j \Rightarrow i \leq j \\ \quad | \text{PatternNotFound} \rightarrow \neg \exists i. MPOS\ i \\ \quad | _ \rightarrow \perp] \end{aligned}$$

where

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

($MPOS\ i$ means that the pattern occurs at the i -th position)

Simplified Hoare rules for Java/JML

$$\frac{\begin{array}{l} \{\phi\} x \leftarrow b; p \{x \Rightarrow \psi \parallel x \Rightarrow S\} \\ \{\phi\} x \leftarrow b; q \{\neg x \Rightarrow \psi \parallel \neg x \Rightarrow S\} \end{array}}{\{\phi\} \text{if } b \text{ then } p \text{ else } q \{\psi \parallel S\}}$$

$$\left(\begin{array}{lcl} \text{diverges} & = & \lambda x . d \\ \text{requires} & = & \textit{Pre} \\ \text{statement} & = & x = b; p \\ \text{ensures} & = & x \Rightarrow \textit{Post} \\ \text{signals} & = & P_{exec} \\ \text{return} & = & P_{ret} \\ \text{break} & = & P_{brk} \\ \text{continue} & = & P_{cnt} \end{array} \right) \quad \left(\begin{array}{lcl} \text{diverges} & = & \lambda x . d \\ \text{requires} & = & \textit{Pre} \\ \text{statement} & = & x = b; q \\ \text{ensures} & = & \neg x \Rightarrow \textit{Post} \\ \text{signals} & = & P_{exec} \\ \text{return} & = & P_{ret} \\ \text{break} & = & P_{brk} \\ \text{continue} & = & P_{cnt} \end{array} \right)$$

$$\left(\begin{array}{lcl} \text{diverges} & = & \lambda x . d \\ \text{requires} & = & \textit{Pre} \\ \text{statement} & = & \textit{if } b \textit{ then } p \textit{ else } q \\ \text{ensures} & = & Q \\ \text{signals} & = & P_{exec} \\ \text{return} & = & P_{ret} \\ \text{break} & = & P_{brk} \\ \text{continue} & = & P_{cnt} \end{array} \right)$$

The resumption monad and parallel composition

Modelling concurrency (tentative!)

Assume a base monad M with nondeterministic choice \square and *fail*.

$$RA = \mu\gamma. M(A + \gamma) \quad (\text{resumptions})$$

A **resumption** makes a step in the base monad, and then

- either stops and delivers a value in A ,
- or continues with a new resumption.

$$\begin{aligned} \text{step} : MA &\rightarrow RA \\ \text{step}(p : MA) &= M(\text{inl})(p) \end{aligned} \quad (\text{make a one-step computation})$$

$$\begin{aligned} \text{run} : RA &\rightarrow MA \\ \text{run}(r : RA) &= \text{case } r \text{ of} \\ &\quad \text{inl}(a) \rightarrow \text{ret } a \\ &\quad \text{inr}(r') \rightarrow \text{run}(r') \end{aligned} \quad (\text{collapse a computation into one step})$$

Dynamic Logic for the Resumption Monad

Basically, the resumption monad **inherits dsef formulas** and dynamic logic from the underlying monad.

This means that with dynamic logic, we can **only** capture the **input/output behaviour** of a program. We cannot capture its potential interaction with other programs during parallel composition.

Hence, we need special proof rules for **parallel composition**.

Parallel composition

$combine : A + RA \rightarrow RA$

$combine(inl(a)) = ret\ a$

$combine(inr(r)) = r$

$- || - : RA \times RB \rightarrow R(A \times B)$

$do\ p' \leftarrow p; ret\ (inr(combine(p') || q))$

$p || q = \square\ do\ q' \leftarrow q; ret\ (inr(p' || combine(q')))$

$\square\ do\ inl(a) \leftarrow p; inl(b) \leftarrow q; ret\ (inl(a, b))$

(*fail* is used in case of failure of pattern-matching)

Proof Rule for Parallel Composition

$$\begin{array}{c}
 \phi_1(p) \\
 \phi_2(q) \\
 \forall r. \{ \phi_1(r) \wedge \psi \} p' \leftarrow r \{ \phi_1(\mathit{combine}(p')) \wedge \psi \} \\
 \forall r. \{ \phi_2(r) \wedge \psi \} q' \leftarrow r \{ \phi_2(\mathit{combine}(q')) \wedge \psi \} \\
 \{ \psi \} \mathit{inl}(a) \leftarrow p; \mathit{inl}(b) \leftarrow q \{ \chi(a, b) \} \\
 \hline
 \{ \psi \} x \leftarrow p \parallel q \{ \chi(x) \}
 \end{array}$$

(||)

Conclusion

- Monads are as useful in specification as in programming
- Have soundly interpreted
 - **Hoare logic** over an arbitrary monad
 - **Dynamic logic** over a not quite as arbitrary monad (but still over most of them)
- Specific monads can be **naturally axiomatized** in computational logics
- Monads can be **combined** by just merging their axiomatizations
- Hoare logic and dynamic logic have been **coded in Isabelle** (verified pattern-matching, Russian multiplication, breadth-first-search)

Open Questions

- better logic for **concurrent programs**, based on traces or transition systems?
- formal underpinning of **concurrent JML**
- useful framework for **monad combination**
- completeness of calculi for **non-simple** monads
- calculus for deterministic **side-effect freeness**

Outlook: a logic for arrows

- **Arrows** (Hughes 2000) are a generalization of monads, taking into account both **input and output** values of computations
- Yampa: **domain-specific embedded languages** for the programming of hybrid reactive systems
- currently used in a student project to specify control of a **wheelchair**

