Monads and Effects

Eugenio Moggi (DISI, Univ. of Genova, Italy)
http://www.disi.unige.it/person/MoggiE/

Monads and computational types:
- simple examples
- mathematical definitions

Metalanguages with computational types

Monadic approach to denotational semantics:
- CBN translation (for Haskell)
- CBV translation (for SML)
- PCF translation (for Algol)

Monad transformers and incremental approach:
- general methodology
- examples

Warning: computational types $\neq$ monads

- Monad/Triple T mathematical notion (Category Theory)

  monads arise in many contexts (e.g. algebraic theories)

  Kleisli and Eilenberg-Moore: monads/comonads $\Longleftrightarrow$ adjunctions

- computational type/notion of computation T informal concept

  $c \in TA \overset{\Delta}{\Longleftrightarrow}$ "c program computing values in A"
  $Y\colon (TA \to TA) \to TA$ recursive definitions of programs

  thesis [Mog91]: notions of computation are (strong) monads

* bulk/collection types T informal concept

  $c \in TA \overset{\Delta}{\Longleftrightarrow}$ "c finite collection of elements of A"
  $0\colon TA$ empty collection and $+\colon TA \to TA \to TA$ merge of collections

  thesis [BNTW95,Man98]: collection types are (strong) monads

Simple examples of notions of computation in sets (no recursive definitions of programs)

- **partiality** $TA = A_\perp \triangleq A + \{\perp\}$, $\perp$ *diverging computation*

- **nondeterminism** $TA = \mathcal{P}_{fin}(A)$ set of finite subsets of A

- **side-effects** $TA = (A \times S)^S$, S is a set of states, e.g. U^L or U^*

- **exceptions** $TA = A + E$, E set of exceptions

- **continuations** $TA = R^{(R^A)}$, R set of results

- **interactive input** $TA = \mu X.A + X^U$, i.e. set of U-branching trees with A-labelled leaves, where $\mu X.\tau$ least solution to domain equation $X = \tau$

- **interactive output** $TA = \mu X.A + (U \times X) \cong U^* \times A$

Simple examples of notions of computation in cpos (allow recursive definitions of programs)

- **partiality** $TA = A_\perp$, i.e. *lifting*

- **nondeterminism** $TA =$ *powerdomain*

- **side-effects** $TA = (A \times S)_\perp^S$

- **exceptions** $TA = (A + E)_\perp$

- **continuations** $TA = R^{(R^A)}$, R cpo with $\perp$

- **interactive input** $TA = \mu X.(A + X^U)_\perp$,
  where $\mu X.\tau$ least solution to domain equation $X = \tau$

- **interactive output** $TA = \mu X.(A + (U \times X))_\perp$

More complex examples of notions of computation in sets

- $TA = ((A + E) \times S)^S$ imperative programs with exceptions

- $TA = ((A \times S) + E)^S$ imperative programs with exceptions

- $TA = \mu X.\mathcal{P}_{fin}(A + (U \times X))$ nondeterministic interactive programs, related to synchronization trees up to strong bisimulation

- $TA = \mu X.\mathcal{P}_{fin}((A + X) \times S)^S$ parallel imperative programs, related to resumptions and small-step operational semantics

Fix a category $\mathcal{C}$, e.g. the category of sets or cpos

$(T, \eta, \_^*)$ **Kleisli triple/triple in extension form** $\overset{\Delta}{\Longleftrightarrow}$

$T\colon |\mathcal{C}| \to |\mathcal{C}|$, $\quad \eta_A\colon A \to TA$ for $A \in |\mathcal{C}|$, $\quad f^*\colon TA \to TB$ for $f\colon A \to TB$, s.t.

- $\eta_A^* = \mathsf{id}_{TA}$ and

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "f"'] & TA \arrow[d, "f^*"] \arrow[ddr, "(f;g^*)^*"] & \\
& TB \arrow[r, "g^*"] & TC
\end{tikzcd}$$

Intuitive justification in terms of computational types:
- TA type of *computations*
- $a\colon A \overset{\eta_A}{\longmapsto} [a]\colon TA$ *inclusion* of values into computations
- $\dfrac{a\colon A \overset{f}{\longmapsto} f\,a\colon TB}{c\colon TA \overset{f^*}{\longmapsto} \mathsf{let}\, a \Leftarrow c\, \mathsf{in}\, f\,a\colon TB}$ *extension* of f to computations

Warning: the first choice to make is which category $\mathcal{C}$ to use, e.g. to support recursive programs or recursive/polymorphic types.

Exercise: examples of notions of computation revised

Extend notions of computation given in sets/cpos to Kleisli triples, e.g.

- **nondeterminism** $TA = \mathcal{P}_{fin}(A)$
  $\eta_A(a) = \{a\}$ and $f^* c = \cup\{f\,x | x \in c\}$
  variant $TA = A^*$ set of finite sequences of A

- **side-effects** $TA = (A \times S)^S$
  $\eta_A(a) = \lambda s{:}\,S.(a, s)$ and $f^* c = \lambda s{:}\,S.\mathsf{let}\,(a, s') = c\,s\,\mathsf{in}\,f\,a\,s'$

- **exceptions** $TA = A + E$
  $\eta_A(a) = \mathsf{inl}\,a$ and $f^* c = \begin{cases} \mathsf{inr}\,e & \text{if } c = \mathsf{inr}\,e \\ f\,a & \text{if } c = \mathsf{inl}\,a \end{cases}$

- **continuations** $TA = R^{(R^A)}$
  $\eta_A(a) = \lambda k{:}\,R^A.k\,a$ and $f^* c = \lambda k{:}\,R^B.c(\lambda a{:}\,A.f\,a\,k)$
  variant $TA = \forall R.R^{(R^A)}$ definable in 2nd-order λ-calculus

The Kleisli category $\mathcal{C}_T$ of programs

- $|\mathcal{C}_T| \triangleq |\mathcal{C}|$

- $\mathcal{C}_T(A, B) \triangleq \mathcal{C}(A, TB)$, programs of type B parameterized w.r.t. A

- $f ;_T g \triangleq f ; g^*$, sequential composition of programs

$$\frac{x\colon A \xmapsto{f} f\,x\colon TB \qquad y\colon B \xmapsto{g} g\,y\colon TC}{x\colon A \xmapsto{f;g^*} \mathsf{let}\; y \Leftarrow f\,x \;\mathsf{in}\; g\,y\colon TC}$$

Prop. Axioms for Kleisli triples amounts to say $\mathcal{C}_T$ is a category

(T, η, μ) monad/triple in monoid form $\overset{\Delta}{\Longleftrightarrow}$

$T\colon \mathcal{C} \to \mathcal{C}$ functor, $\eta\colon \mathrm{id}_{\mathcal{C}} \dot{\to} T$ and $\mu\colon T^2 \dot{\to} T$ natural transformations, s.t.

$$\begin{tikzcd} T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\ T^2A \arrow[r, "\mu_A"] & TA \end{tikzcd} \qquad \begin{tikzcd} TA \arrow[r, "\eta_{TA}"] \arrow[dr, "\mathrm{id}_{TA}"'] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "\mathrm{id}_{TA}"] \\ & TA & \end{tikzcd}$$

Prop. There is a bijection between Kleisli triples and monads:

- $(f\colon A \to TB)^* \triangleq TA \xrightarrow{Tf} T^2B \xrightarrow{\mu_B} TB$

- $T(f\colon A \to B) \triangleq (A \xrightarrow{f} B \xrightarrow{\eta_B} TB)^*$
- $\mu_A \triangleq (TA \xrightarrow{\mathrm{id}_{TA}} TA)^*$

Warning: to verify that "(T, η, μ) is a monad" there are 7 equations to check!

The Eilenberg-Moore category $\mathcal{C}^T$ of T-algebras

- $\alpha \in |\mathcal{C}^T| \overset{\Delta}{\Longleftrightarrow}$

$$\begin{tikzcd} T^2A \arrow[r, "\mu_A"] \arrow[d, "T\alpha"'] & TA \arrow[d, "\alpha"] \\ TA \arrow[r, "\alpha"'] & A \end{tikzcd}$$

$$\begin{tikzcd} A \arrow[r, "\eta_A"] \arrow[dr, "\mathsf{id}_A"'] & TA \arrow[d, "\alpha"] \\ & A \end{tikzcd}$$

- $f \in \mathcal{C}^T(\alpha, \beta) \overset{\Delta}{\Longleftrightarrow}$

$$\begin{tikzcd} TA \arrow[r, "Tf"] \arrow[d, "\alpha"'] & TB \arrow[d, "\beta"] \\ TA \arrow[r, "f"'] & B \end{tikzcd}$$

- $f ;^T g \overset{\Delta}{=} f ; g$

Monad on sets induced by algebraic signature Σ

- Σ single-sorted algebraic signature

- $TX \triangleq T_\Sigma(X)$ set of Σ-terms with variables in X
- $\eta_X\, x \triangleq x$, i.e. variable x as a term
- $f^*\, t \triangleq t[f]$, i.e. substituting in term t every variable $x \in X$ with term $f\, x$

Prop. Eilenberg-Moore category $\mathcal{C}^T$ isomorphic to category of Σ-algebras

The correspondence extends to single-sorted algebraic theories

Relations between $\mathcal{C}$, $\mathcal{C}_T$ and $\mathcal{C}^T$

$$\begin{tikzcd}
\mathcal{C} & \mathcal{C}^T \arrow[l, "U"'] & T\text{-algebras} \\
 & \mathcal{C}_T \arrow[ul, "F"] \arrow[u, hook, "\Phi"'] & \text{free } T\text{-algebras}
\end{tikzcd}$$

- $U(\alpha\colon TA \to A) \triangleq A$, the carrier of T-algebra α
- $FA \triangleq A$
- $\Phi A \triangleq \mu_A\colon T^2A \to TA$, the free T-algebra over A

Prop. $F;\Phi \dashv U$ and $F \dashv \Phi;U$

Prop. Both adjunctions induce the monad T

The category $Mon(\mathcal{C})$ of monads/Kleisli triples over $\mathcal{C}$

- $|Mon(\mathcal{C})| \triangleq$ monads/Kleisli triples over $\mathcal{C}$

- $\sigma\colon (T, \eta, \_^*) \to (T', \eta', \_^{*'}) \overset{\Delta}{\Longleftrightarrow} \sigma_A\colon TA \to T'A$ for $A \in |\mathcal{C}|$ s.t.
1) $\eta_A; \sigma_A = \eta'_A$
2) $f^*; \sigma_B = \sigma_A; (f; \sigma_B)^{*'}$ for $f\colon A \to TB$

Prop. There is a bijection between monad morphisms $\sigma\colon T \to T'$ and *carrier preserving* functors $V\colon \mathcal{C}^{T'} \to \mathcal{C}^T$, i.e. $U(V\alpha') = U(\alpha'\colon T'A \to A) = A$

Prop. There is a bijection between monad morphisms $\sigma_A\colon TA \to T'A$ and *layerings* $\zeta_A\colon T(T'A) \to T'A$ of T' over T [Fil99], namely $\zeta_A = V\mu_A$

Metalanguages[§] with computational types
(the syntactic/axiomatic counter-part of monads)

(§) e.g. equational logic and higher-order λ-calculus, with judgments:
$\Gamma \vdash$, $\Gamma \vdash \tau\, type$, $\Gamma \vdash e\colon \tau$, $\Gamma \vdash e_1 = e_2\colon \tau$

$$T\ \frac{\Gamma \vdash \tau\, type}{\Gamma \vdash T\tau\, type} \qquad \mathsf{lift}\ \frac{\Gamma \vdash e\colon \tau}{\Gamma \vdash [e]_T\colon T\tau} \qquad \mathsf{let}\ \frac{\Gamma \vdash e_1\colon T\tau_1 \quad \Gamma, x\colon \tau_1 \vdash e_2\colon T\tau_2}{\Gamma \vdash (\mathsf{let}_T\, x{\Leftarrow}e_1\, \mathsf{in}\, e_2)\colon T\tau_2}\ *$$

$$\frac{\Gamma \vdash e_1\colon T\tau_1 \quad \Gamma, x_1\colon \tau_1 \vdash e_2\colon T\tau_2 \quad \Gamma, x_2\colon \tau_2 \vdash e_3\colon T\tau_3}{\Gamma \vdash \mathsf{let}_T\, x_2{\Leftarrow}(\mathsf{let}_T\, x_1{\Leftarrow}e_1\, \mathsf{in}\, e_2)\, \mathsf{in}\, e_3 = \mathsf{let}_T\, x_1{\Leftarrow}e_1\, \mathsf{in}\, (\mathsf{let}_T\, x_2{\Leftarrow}e_2\, \mathsf{in}\, e_3)\colon T\tau_3}$$

$$\frac{\Gamma \vdash e_1\colon \tau_1 \quad \Gamma, x\colon \tau_1 \vdash e_2\colon T\tau_2}{\Gamma \vdash \mathsf{let}_T\, x{\Leftarrow}[e_1]_T\, \mathsf{in}\, e_2 = e_2[x\colon = e_1]\colon T\tau_2} \qquad \frac{\Gamma \vdash e\colon T\tau}{\Gamma \vdash \mathsf{let}_T\, x{\Leftarrow}e\, \mathsf{in}\, [x]_T = e\colon T\tau}$$

Prop. The axioms are sound and complete w.r.t. interpretation in a category with a parameterized[§] monad (see [Mog91] for details)

(§) parameterization related to the nature of contexts Γ, need of parametrized notions by now well-understood in categorical logic (see [Jac99,Pit00])

Interpretation in a category with finite products

General pattern for interpreting a simply typed calculus:
- context $\Gamma \vdash$ and type $\vdash \tau\, type$ interpreted by objects Γ and τ;
- term $\Gamma \vdash e\colon \tau$ interpreted by morphism $f\colon \Gamma \to \tau$

RULE	SYNTAX		SEMANTICS
T			
	$\vdash \tau\, type$	$=$	τ
	$\vdash T\tau\, type$	$=$	$T\tau$
lift			
	$\Gamma \vdash e\colon \tau$	$=$	$f\colon \Gamma \to \tau$
	$\Gamma \vdash [e]_T\colon T\tau$	$=$	$f; \eta_\tau\colon \Gamma \to T\tau$
let			
	$\Gamma \vdash e_1\colon T\tau_1$	$=$	$f_1\colon \Gamma \to T\tau_1$
	$\Gamma, x\colon \tau_1 \vdash e_2\colon T\tau_2$	$=$	$f_2\colon \Gamma \times \tau_1 \to T\tau_2$
	$\Gamma \vdash \mathsf{let}_T\, x \Leftarrow e_1\, \mathsf{in}\, e_2\colon T\tau_2$	$=$	$(\mathsf{id}_\Gamma, f_1); f_2^*\colon \Gamma \to T\tau_2$

parameterized extension operation $\_^*$ maps $f\colon C \times A \to TB$ to $f^*\colon C \times TA \to TB$

Syntactic sugar

- iterated-let: $\mathsf{let}_T\, \emptyset \Leftarrow \emptyset\, \mathsf{in}\, e' \triangleq e'$, $\mathsf{let}_T\, x, \overline{x} \Leftarrow e, \overline{e}\, \mathsf{in}\, e' \triangleq \mathsf{let}_T\, x \Leftarrow e\, \mathsf{in}\, \mathsf{let}_T\, \overline{x} \Leftarrow \overline{e}\, \mathsf{in}\, e'$
- do-notation: $\mathsf{do}\{e'\} \triangleq e'$, $\mathsf{do}\{x \Leftarrow e;\ stmts\} \triangleq \mathsf{let}_T\, x \Leftarrow e\, \mathsf{in}\, \mathsf{do}\{stmts\}$

Combinators definable in higher order λ-calculus

- triple in extension form:

$$
\begin{array}{l}
T\colon \bullet \to \bullet \\
\hline
unit_T\colon \forall X\colon \bullet . X \to TX \\
unit_T\, X\, x = [x]_T \\
\hline
let_T\colon \forall X, Y\colon \bullet . (X \to TY) \to TX \to TY \\
let_T\, X\, Y\, f\, c = \mathsf{let}_T\, x \Leftarrow c\, \mathsf{in}\, f\, x
\end{array}
$$

- triple in monoid form: T, $unit_T$ and

$$
\begin{array}{l}
map_T\colon \forall X, Y\colon \bullet . (X \to Y) \to TX \to TY \\
map_T\, X\, Y\, f\, c = \mathsf{let}_T\, x \Leftarrow c\, \mathsf{in}\, [f\, x]_T \\
\hline
flat_T\colon \forall X\colon \bullet . T^2 X \to TX \\
flat_T\, X\, c = \mathsf{let}_T\, x \Leftarrow c\, \mathsf{in}\, x
\end{array}
$$

Definitions in metalanguage with computational types

- $\alpha\colon TA \to A$ T-algebra $\overset{\Delta}{\Longleftrightarrow}$

$$\begin{array}{rcl} x\colon A & \vdash & \alpha\,[x]_T = x\colon A \\ c\colon T^2 A & \vdash & \alpha(\mathsf{let}_T\, x{\Leftarrow}c\,\mathsf{in}\,x) = \alpha(\mathsf{let}_T\, x{\Leftarrow}c\,\mathsf{in}\,[\alpha\, x]_T)\colon A \end{array}$$

- $f\colon A \to B$ T-algebra morphism from $\alpha\colon TA \to A$ to $\beta\colon TB \to B$ $\overset{\Delta}{\Longleftrightarrow}$

$$c\colon TA \;\vdash\; f(\alpha\, c) = \beta(\mathsf{let}_T\, x{\Leftarrow}c\,\mathsf{in}\,[f x]_T)\colon B$$

- $\sigma\colon \forall X\colon \bullet . TX \to T'X$ monad morphism from T to T' $\overset{\Delta}{\Longleftrightarrow}$

$$\begin{array}{rcl} X\colon \bullet, x\colon X & \vdash & \sigma\, X\, [x]_T = [x]_{T'}\colon T'X \\ X, Y\colon \bullet, c\colon TX, f\colon X \to TY & \vdash & \sigma\, Y\, (\mathsf{let}_T\, x{\Leftarrow}c\,\mathsf{in}\, f\, x) = \\ & & \mathsf{let}_{T'}\, x{\Leftarrow}\sigma\, X\, c\,\mathsf{in}\,\sigma\, Y\, (f\, x)\colon T'Y \end{array}$$

Monadic approach to denotational semantics

$$PL \xrightarrow{transl} ML_T(\Sigma) \xrightarrow{interp} \mathcal{C}$$

1) identify metalanguage $ML_T(\Sigma)$ with comp. types: auxiliary notation;
2) define translation of programming language PL into $ML_T(\Sigma)$;
3) construct model of $ML_T(\Sigma)$, e.g. via translation.

Pros and Cons

+ translation $PL \to ML_T(\Sigma)$ simple and usually
  no need to be redefined (only extended) when PL is extended;
+ same $ML_T(\Sigma)$ can be reused for several translations/interpretations;

– interpretation of $ML_T(\Sigma)$ gets complex when PL/monad is complex,
  but monad transformers/incremental approach provide a partial solution.

CBN translation

- pattern: $\{x_i\colon \tau_i | i \in m\} \vdash_{PL} e\colon \tau$ translated to $\{x_i\colon T\tau_i^n | i \in m\} \vdash_{ML} e^n\colon T\tau^n$

- $\Sigma \triangleq$ signature for datatype of integers,
  $Y\colon \forall X\colon \bullet.(TX \to TX) \to TX$ fix-point combinator

- types: $\mathsf{Int}^n \triangleq \mathsf{Int}$ $\quad (\tau_1 \to \tau_2)^n \triangleq T\tau_1^n \to T\tau_2^n$ $\quad (\tau_1 \times \tau_2)^n \triangleq T\tau_1^n \times T\tau_2^n$

$e \in PL$	$e^n \in ML_T(\Sigma)$
x	x
n	$[\mathsf{n}]_T$
$e_0 + e_1$	$\mathsf{let}_T\, x_0, x_1 \Leftarrow e_0^n, e_1^n\, \mathsf{in}\, [x_0 + x_1]_T$
$\mu x\colon \tau.e$	$Y\, \tau^n\, (\lambda x\colon T\tau^n.e^n)$
$\lambda x\colon \tau.e$	$[\lambda x\colon T\tau^n.e^n]_T$
$e_1\, e_2$	$\mathsf{let}_T\, f \Leftarrow e_1^n\, \mathsf{in}\, f\, e_2^n$
(e_1, e_2)	$[(e_1^n, e_2^n)]_T$
$\pi_i\, e$	$\mathsf{let}_T\, x \Leftarrow e^n\, \mathsf{in}\, \pi_i\, x$

CBV translation

- pattern: $\{x_i\colon \tau_i | i \in m\} \vdash_{PL} e\colon \tau$ translated to $\{x_i\colon \tau_i^v | i \in m\} \vdash_{ML} e^v\colon T\tau^v$

- $\Sigma \triangleq$ signature for datatype of integers,
  $Y\colon \forall X\colon \bullet.(TX \to TX) \to TX$ fix-point combinator

- types: $\mathsf{Int}^v \triangleq \mathsf{Int}$ $\quad (\tau_1 \to \tau_2)^v \triangleq \tau_1^v \to T\tau_2^v$ $\quad (\tau_1 \times \tau_2)^v \triangleq \tau_1^v \times \tau_2^v$

$e \in PL$	$e^v \in ML_T(\Sigma)$
x	$[x]_T$
n	$[\mathsf{n}]_T$
$e_0 + e_1$	$\mathsf{let}_T\, x_0, x_1 \Leftarrow e_0^v, e_1^v\, \mathsf{in}\, [x_0 + x_1]_T$
$\mu f\colon \tau_1 \to \tau_2.\lambda x\colon \tau_1.e$	$*Y\ (\tau_1 \to \tau_2)^v\ (\lambda f\colon (\tau_1 \to \tau_2)^v.\lambda x\colon \tau_1^v.e^v)$
$\lambda x\colon \tau.e$	$[\lambda x\colon \tau^v.e^v]_T$
$e_1\, e_2$	$\mathsf{let}_T\, f, x \Leftarrow e_1^v, e_2^v\, \mathsf{in}\, f\, x$
(e_1, e_2)	$\mathsf{let}_T\, x_1, x_2 \Leftarrow e_1^v, e_2^v\, \mathsf{in}\, [(x_1, x_2)]_T$
$\pi_i\, e$	$\mathsf{let}_T\, x \Leftarrow e^v\, \mathsf{in}\, [\pi_i\, x]_T$

Extension of let and fix-point combinators to T-algebras

$\bullet_*$ sort of T-algebras

$U\colon \bullet_* \to \bullet$ forgetful functor

$*let_T\colon \forall X\colon \bullet.\forall \alpha\colon \bullet_*.(X \to U\alpha) \to TX \to U\alpha$

$*Y\colon \forall X\alpha\colon \bullet_*.(U\alpha \to U\alpha) \to U\alpha$

$$*\mathsf{let}\ \frac{\Gamma \vdash e_1\colon T\tau_1 \quad \Gamma, x\colon \tau_1 \vdash e_2\colon U(\alpha\colon T\tau_2 \to \tau_2)}{\Gamma \vdash *\mathsf{let}_T\, x{\Leftarrow}e_1 \,\mathsf{in}\, e_2 \triangleq \alpha(\mathsf{let}_T\, x{\Leftarrow}e_1 \,\mathsf{in}\, [e_2]_T)\colon U(\alpha\colon T\tau_2 \to \tau_2)}$$

$$*\mathsf{Y}\ \frac{\Gamma, x\colon \tau \vdash e\colon U(\alpha\colon T\tau \to \tau)}{\Gamma \vdash *Y\,\tau\,(\lambda x\colon \tau.e) \triangleq \alpha(Y\,\tau\,(\lambda c\colon T\tau.[e[x\colon = \alpha\, c]]_T)\colon U(\alpha\colon T\tau \to \tau)}$$

to prove "$*Y$ fix-point combinator" it suffices that $\eta_A; \alpha = \mathsf{id}_A$

CBV translation extended to PL with integer locations

- $\Sigma \triangleq$ ...,
  Loc set of locations,
  ℓ: Loc fixed set of constants,
  get: Loc $\to T$Int get integer from location,
  set: Loc $\to$ Int $\to T1$ store integer into location
- interpretation of monad T, e.g. side-effects
- types: ... $\quad \mathsf{Loc}^v \triangleq \mathsf{Loc} \quad \mathsf{Unit}^v \triangleq 1$

$e \in PL$	$e^v \in ML_T(\Sigma)$
ℓ	$[\ell]_T$
$!e$	$\mathsf{let}_T\, l \Leftarrow e^v \,\mathsf{in}\, get\, l$
$()$	$[()]_T$
$e_0 := e_1$	$\mathsf{let}_T\, l, n \Leftarrow e_0^v, e_1^v \,\mathsf{in}\, set\, l\, n$

PCF translation

- pattern: $\{x_i\colon \tau_i | i \in m\} \vdash_{PL} e\colon \tau$ translated to $\{x_i\colon \tau_i^a | i \in m\} \vdash_{ML} e^a\colon \tau^a$

- $\Sigma \triangleq$ signature for datatype of integers,
  $Y\colon \forall X\colon \bullet.(TX \to TX) \to TX$ fix-point combinator

- types: $\mathsf{Int}^a \triangleq T\mathsf{Int}$ $\quad (\tau_1 \to \tau_2)^a \triangleq \tau_1^a \to \tau_2^a$ $\quad (\tau_1 \times \tau_2)^a \triangleq \tau_1^a \times \tau_2^a$

$e \in PL$	$e^a \in ML_T(\Sigma)$
x	x
n	$[\mathsf{n}]_T$
$e_0 + e_1$	$\mathsf{let}_T\, x_0, x_1 \Leftarrow e_0^a, e_1^a\ \mathsf{in}\ [x_0 + x_1]_T$
$\mu x\colon \tau.e$	$*Y\ \tau^a\ (\lambda x\colon \tau^a.e^a)$
$\lambda x\colon \tau.e$	$\lambda x\colon \tau^a.e^a$
$e_1\, e_2$	$e_1^a\, e_2^a$
(e_1, e_2)	(e_1^a, e_2^a)
$\pi_i\, e$	$\pi_i\, e^a$

PCF translation extended to Algol

- monads $T \xrightarrow{\sigma} T'$,e.g. T state-readers and T' side-effects
- $\Sigma \triangleq$ $\ldots,$
  Loc set of locations,
  ℓ: Loc fixed set of constants,
  get: Loc $\to T$Int get integer from location,
  set: Loc $\to$ Int $\to T'1$ store integer into location
- types: $\ldots$ $\quad \mathsf{Loc}^a \triangleq T\mathsf{Loc} \quad \mathsf{Cmd}^a \triangleq T'1$

$e \in PL$	$e^a \in ML_{T,T'}(\Sigma)$
ℓ	$[\ell]_T$
$!e$	$\mathsf{let}_T\, l \Leftarrow e^a \,\mathsf{in}\, get\, l$
$e_0 := e_1$	$\mathsf{let}_{T'}\, l, n \Leftarrow \sigma e_0^a, \sigma e_1^a \,\mathsf{in}\, set\, l\, n$
skip	$[()]_{T'}$
$e_0; e_1$	$\mathsf{let}_{T'}\, \_ \Leftarrow e_0^a \,\mathsf{in}\, e_1^a$

Incremental approach: general methodology

$$PL \xrightarrow{transl} ML(\Sigma_n) \xrightarrow{transl} \ldots \xrightarrow{transl} ML(\Sigma_0) \xrightarrow{interp} \mathcal{C}$$

incremental definition of auxiliary notation [CM93,Mog97,LHJ95,LH96,Fil99]

Filinski's approach: $I\colon ML_{T',\overline{T}}(\Sigma_{old} + \Sigma'_{new}) \to ML_{\overline{T}}(\Sigma_{old})$

1) I definitional extension, i.e. identity on $ML_{\overline{T}}(\Sigma_{old})$

2) T' new computational type, monadic reflection and reification

3) Σ'_{new} auxiliary notation related to T'

Usually T' defined by applying monad transformer for adding new computational effect to a T in $\overline{T}$. Σ'_{new} may include operations op' extending to T' pre-existing op for T, and operations associated to new computational effect.

Monad transformers semantically

- function $F\colon |Mon(\mathcal{C})| \to |Mon(\mathcal{C})|$ mapping monads (over $\mathcal{C}$) to monads
- monad morphism $in_T\colon T \to FT$ for any monad T

Often F functor on $Mon(\mathcal{C})$, and $in\colon \mathrm{id}_{Mon(\mathcal{C})} \dot{\to} F$ natural transformation

Monad transformers syntactically: $I_F\colon ML_{T',T}(\Sigma_{par}) \to ML_T(\Sigma_{par})$

- Σ_{par} parameters of the monad transformer

Examples of I_F in higher order λ-calculus: general pattern

- new monad T' and monad morphism $in\colon T \to T'$
- operations in Σ_{new} associated to new computational effect
- operation $op'\colon \forall X\colon \bullet.A \to (B \to T'X) \to T'X$ extending
  pre-existing $op\colon \forall X\colon \bullet.A \to (B \to TX) \to TX$

Haskell constructor classes provide convenient setting for the incremental approach [LHJ95]: type inference allows concise and readable definitions, type-checking detects most errors.

Logical frameworks needed to express and validate properties [Mog97].

Monad transformer I_{se} for adding side-effects

Σ_{par} states $S\colon \bullet$

Σ_{new} lookup $lkp'\colon T'S$
update $upd'\colon S \to T'1$

$$
\begin{array}{rcl}
T'X & \triangleq & S \to T(X \times S) \\
[x]_{T'} & \triangleq & \lambda s.[(x,s)]_T \\
\mathsf{let}_{T'}\, x \Leftarrow c\, \mathsf{in}\, f\, x & \triangleq & \lambda s.\mathsf{let}_T\, (x,s') \Leftarrow c\, s\, \mathsf{in}\, f\, x\, s' \\
in\, X\, c & \triangleq & \lambda s.\mathsf{let}_T\, x \Leftarrow c\, \mathsf{in}\, [(x,s)]_T \\
\hline
lkp' & \triangleq & \lambda s.[(s,s)]_T \\
upd'\, s & \triangleq & \lambda s'.[(*,s)]_T \\
\hline
op'\, X\, a\, f & \triangleq & \lambda s.op\, (X \times S)\, a\, (\lambda b.f\, b\, s)
\end{array}
$$

lkp' and upd' do not fit format for op. However, given $*op\colon A \to TB$ one can define $op\colon \forall X\colon \bullet.A \to (B \to TX) \to TX$, i.e. $op\, X\, a\, f \triangleq \mathsf{let}_T\, b \Leftarrow *op\, a\, \mathsf{in}\, f\, b$

Monad transformer I_{ex} for adding exceptions

Σ_{par} exceptions $E\colon \bullet$

Σ_{new} raise $raise'\colon \forall X\colon \bullet . E \to T'X$
handle $handle'\colon \forall X\colon \bullet . (E \to T'X) \to T'X \to T'X$

$$
\begin{array}{rcl}
T'X & \triangleq & T(X+E) \\
[x]_{T'} & \triangleq & [\mathsf{inl}\, x]_T \\
\mathsf{let}_{T'}\, x \Leftarrow c\, \mathsf{in}\, f\, x & \triangleq & \mathsf{let}_T\, u \Leftarrow c\, \mathsf{in}\, (\mathsf{case}\, u\, \mathsf{of}\, x \Rightarrow f\, x \mid n \Rightarrow [\mathsf{inr}\, n]_T) \\
in\, X\, c & \triangleq & \mathsf{let}_T\, x \Leftarrow c\, \mathsf{in}\, [\mathsf{inl}\, x]_T \\
\hline
raise'\, X\, n & \triangleq & [\mathsf{inr}\, n]_T \\
handle'\, X\, f\, c & \triangleq & \mathsf{let}_T\, u \Leftarrow c\, \mathsf{in}\, (\mathsf{case}\, u\, \mathsf{of}\, x \Rightarrow [\mathsf{inl}\, x]_T \mid n \Rightarrow f\, n) \\
\hline
op'\, X\, a\, f & \triangleq & op\, (X+E)\, a\, f
\end{array}
$$

The definition of op' works for extending more general type of operations

Monad transformer I_{co} for adding complexity

Σ_{par} monoid $M\colon \bullet$
$1\colon M$
$*\colon M \to M \to M$ (we use infix notation for $*$)

to prove "T' monad" we should add axioms "$(M, 1, *)$ monoid"

Σ_{new} cost $tick'\colon M \to T'1$

$$
\begin{array}{rcl}
T'X & \triangleq & T(X \times M) \\
[x]_{T'} & \triangleq & [(x, 1)]_T \\
\mathsf{let}_{T'}\, x \Leftarrow c\, \mathsf{in}\, f\, x & \triangleq & \mathsf{let}_T\, (x, m) \Leftarrow c\, \mathsf{in}\, (\mathsf{let}_T\, (y, n) \Leftarrow f\, x\, \mathsf{in}\, [(y, m * n)]_T) \\
in\, X\, c & \triangleq & \mathsf{let}_T\, x \Leftarrow c\, \mathsf{in}\, [(x, 1)]_T \\
\hline
tick'\, m & \triangleq & [(*, m)]_T \\
\hline
op'\, X\, a\, f & \triangleq & op\, (X \times M)\, a\, f
\end{array}
$$

Monad transformer I_{con} for adding continuations

Σ_{par} results $R\colon \bullet$

Σ_{new} abort $abort'\colon \forall X\colon \bullet . R \to T'X$
call-cc $callcc'\colon \forall X, Y\colon \bullet . ((X \to T'Y) \to T'X) \to T'X$

$$
\begin{array}{rcl}
T'X & \triangleq & (X \to TR) \to TR \\
[x]_{T'} & \triangleq & \lambda k.k\,x \\
\mathsf{let}_{T'}\, x{\Leftarrow}c\,\mathsf{in}\, f\,x & \triangleq & \lambda k.c\,(\lambda x.f\,x\,k) \\
in\,X\,c & \triangleq & \lambda k.\mathsf{let}_T\, x{\Leftarrow}c\,\mathsf{in}\, k\,x \\
\hline
abort'\,X\,r & \triangleq & \lambda k.[r]_T \\
callcc'\,X\,Y\,f & \triangleq & \lambda k.f\,(\lambda x.\lambda k'.[k\,x]_T)\,k \\
\hline
op'\,X\,a\,f & \triangleq & \lambda k.op\,R\,a\,(\lambda b.f\,b\,k)
\end{array}
$$

$callcc'$ does not fit format for op. I_{con} does not extend to a functor

Exercises

3.3) Extend CBN/CBV/PCF translation to PL with polymorphic types.

3.7) Use translation to Validate laws for PL, or find counterexamples.

3.8) Give translation for Haskell with interger locations.

3.9) Give translation for SML with allocation of interger locations.

3.10) Give translation for SML with references of any type.

3.14) Validate monad transformers, i.e. T' monad and in monad morphism.

3.15) For each monad transformer define Y' for T' given Y for T.
Try to extend an op of type $\forall X, Y\colon \bullet.((X \to TY) \to TX) \to TX$.

3.16) Define monad transformer I_{sr} for state-readers, and relate it to I_{se}.

3.17) Check which monad transformers commute up to iso.

HARD) Define predicates for $c \in T'X \triangleq S \to T(X \times S)$ capturing properties: "c does not read from S" and "c does not write in S".
Define a predicate for $c \in T'X \triangleq T(X + E)$ capturing property "c does not raise exceptions in E".