

## Dynamic Classification

Inspired by the type  $\tau_{exn}$  that arose in our study of exceptions, we define a language  $\mathcal{L}\{\text{classified}\}$  of values classified using dynamically generated symbols (per  $\mathcal{L}\{\text{fluid new}\}$ ) with the following syntax:

Category	Item	Abstract	Concrete
Type	$\tau$	::= <code>clsfd</code>	<code>clsfd</code>
Expr	$e$	::= <code>in[a](e)</code>   <code>ccase(e; e<sub>0</sub>; r<sub>1</sub>, ..., r<sub>n</sub>)</code>	<code>in[a](e)</code> <code>ccase e{r<sub>1</sub>   ...   r<sub>n</sub>} ow e<sub>0</sub></code>
Rule	$r$	::= <code>in?[a](x.e)</code>	<code>in[a](x) ⇒ e</code>

1

### Typing Rules for $\mathcal{L}\{\text{classified}\}$

Judgement  $\Sigma \Gamma r : \tau$ , where  $\Sigma$  specifies types of symbols (as in  $\mathcal{L}\{\text{fluid new}\}$ ), says rule  $r$  matches a classified value of form `in[a](e)` and yields a value of type  $\tau$ .

$$\frac{\frac{\Sigma \vdash a : \tau \quad \Sigma \Gamma \vdash e : \tau}{\Sigma \Gamma \vdash \text{in}[a](e) : \text{clsfd}} \quad \frac{\Sigma \vdash e : \text{clsfd} \quad \Sigma \Gamma \vdash e_0 : \tau \quad \Sigma \Gamma \vdash r_1 : \tau \quad \dots \quad \Sigma \Gamma \vdash r_n : \tau}{\Sigma \Gamma \vdash \text{ccase}(e; e_0; r_1, \dots, r_n) : \tau}}{\Sigma \vdash a : \sigma \quad \Sigma \Gamma, x : \sigma \vdash e : \tau}{\Sigma \Gamma \vdash \text{in}^?[a](x.e) : \tau}$$

2

### Dynamic Semantics for $\mathcal{L}\{\text{classified}\}$

Using same abstract machine as for  $\mathcal{L}\{\text{fluid new}\}$ :

$$\frac{\frac{\frac{e \text{ val}}{\text{in}[a](e) \text{ val}}}{e @ \nu \mapsto e_0 @ \nu'}}{\text{in}[a](e) @ \nu \mapsto \text{in}[a](e_0) @ \nu'}}{\frac{e @ \nu \mapsto e' @ \nu'}{\text{ccase}(e; e_0; r_1, \dots, r_n) @ \nu \mapsto \text{ccase}(e'; e_0; r_1, \dots, r_n) @ \nu'}}$$

3

### Dynamic Semantics for $\mathcal{L}\{\text{classified}\}$

$$\frac{\frac{\text{in}[a](e) \text{ val}}{\text{ccase}(\text{in}[a](e); e_0; e) @ \nu \mapsto e_0 @ \nu}}{\text{ccase}(\text{in}[a_1](e_1); e_0; \text{in}^?[a_1](x_1.e'_1), \dots, \text{in}^?[a_n](x_n.e'_n)) @ \nu \mapsto [e_1/x_1]e'_1 @ \nu}}{\frac{\text{in}[a](e) \text{ val} \quad a \neq a_1 \quad n > 0}{\text{ccase}(\text{in}[a](e); e_0; \text{in}^?[a_1](x_1.e'_1), \dots, \text{in}^?[a_n](x_n.e'_n)) @ \nu \mapsto \text{ccase}(\text{in}[a](e); e_0; \text{in}^?[a_2](x_2.e'_2), \dots, \text{in}^?[a_n](x_n.e'_n)) @ \nu}}$$

4

### Preservation for $\mathcal{L}\{\text{classified}\}$

#### Theorem 1 (Preservation)

Suppose that  $e @ \nu \mapsto e' @ \nu'$ , where  $\Sigma \vdash e : \tau$  and  $\Sigma \vdash a : \tau_a$  whenever  $a \in \nu$ . Then  $\nu' \supseteq \nu$ , and there exists  $\Sigma' \supseteq \Sigma$  such that  $\Sigma' \vdash e' : \tau$  and  $\Sigma' \vdash a' : \tau_{a'}$  for each  $a' \in \nu'$

Proof is by induction on evaluation.

5

### Progress for $\mathcal{L}\{\text{classified}\}$

The canonical forms lemma characterizes closed values of  $\mathcal{L}\{\text{classified}\}$ :

#### Lemma 2

Suppose that  $\Sigma \vdash e : \text{clsfd}$  and  $e$  val. Then  $e = \text{in}[a](e')$  for some  $a$  such that  $\Sigma \vdash a : \tau$  and some  $e'$  such that  $e'$  val and  $\Sigma \vdash e' : \tau$ .

#### Theorem 3 (Progress)

Suppose that  $\Sigma \vdash e : \tau$  and that if  $a \in \nu$ , then  $\Sigma \vdash a : \tau_a$  for some type  $\tau_a$ . Then either  $e$  val or  $e @ \nu \mapsto e' @ \nu'$  for some  $\nu'$  and  $e'$ .

6

### Dynamic Classes

To support direct association of dynamically generated symbols with the labels used in data classification we define a language  $\mathcal{L}\{\text{class}\}$  that defines a new kind of type to be combined with the dynamic symbol generation capability of  $\mathcal{L}\{\text{fluid new}\}$  using the following syntax:

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{class}(\tau)$	$\tau \text{ class}$
Expr	$e$	$::= \text{cls}[a]$   $\text{ccase}[\tau, \sigma](e; e_0; r_1, \dots, r_n)$	$\text{cls}[a]$ $\text{ccase } e\{r_1 \mid \dots \mid r_n\} \text{ow } e_0$
Rule	$r$	$::= \text{cls?}[a](e)$	$\text{cls}[a] \Rightarrow e$

7

### Typing Rules for $\mathcal{L}\{\text{class}\}$

$$\frac{\Sigma \vdash a : \tau}{\Sigma \Gamma \vdash \text{cls}[a] : \text{class}(\tau)}$$

$$\frac{\Sigma \Gamma \vdash e : \text{class}(\tau) \quad \Sigma \Gamma \vdash e_0 : [\tau/t]\sigma \quad \Sigma \Gamma \vdash r_1 : \text{class}(\tau_1) > [\tau_1/t]\sigma \quad \dots \quad \Sigma \Gamma \vdash r_n : \text{class}(\tau_n) > [\tau_n/t]\sigma}{\Sigma \Gamma \vdash \text{ccase}[\tau, \sigma](e; e_0; r_1, \dots, r_n) : [\tau/t]\sigma}$$

$$\frac{\Sigma \vdash a : \tau \quad \Sigma \Gamma \vdash e : \tau'}{\Sigma \Gamma \vdash \text{cls?}[a](e) : \text{class}(\tau) > \tau'}$$

Note how the use of parametric typing propagates type identity information gained during pattern matching.

8

### Preservation for $\mathcal{L}\{\text{class}\}$

Dynamic semantics for  $\mathcal{L}\{\text{class}\}$  similar to those for  $\mathcal{L}\{\text{classified}\}$ , again using same abstract machine as for  $\mathcal{L}\{\text{fluid new}\}$ .

#### Theorem 4 (Preservation)

Suppose that  $e @ \nu \mapsto e' @ \nu'$ , where  $\Sigma \vdash e : \tau$  and  $\Sigma \vdash a : \tau_a$  whenever  $a \in \nu$ . Then  $\nu' \supseteq \nu$ , and there exists  $\Sigma' \supseteq \Sigma$  such that  $\Sigma' \vdash e' : \tau$  and  $\Sigma' \vdash a' : \tau_a$  for each  $a' \in \nu'$ .

Proof is by induction on evaluation.

9

### Progress for $\mathcal{L}\{\text{class}\}$

The canonical forms lemma characterizes closed values of  $\mathcal{L}\{\text{class}\}$ :

#### Lemma 5

Suppose that  $\Sigma \vdash e : \tau \text{ class}$  and  $e$  val. Then  $e = \text{cls}[a]$  for some  $a$  such that  $\Sigma \vdash a : \tau$ .

#### Theorem 6 (Progress)

Suppose that  $\Sigma \vdash e : \tau$  and that if  $a \in \nu$ , then  $\Sigma \vdash a : \tau_a$  for some type  $\tau_a$ . Then either  $e$  val or  $e @ \nu \mapsto e' @ \nu'$  for some  $\nu'$  and  $e'$ .

Dynamic classification ( $\mathcal{L}\{\text{classified}\}$ ) can be defined using dynamic classes ( $\mathcal{L}\{\text{class}\}$ ), existential types and product types. See Harper for details.

10

### Computational Effects

There has long been interest in segregating expressions that use **computational effects** – constraints on order of execution beyond those imposed by data flow – from expressions that don't.

One approach: distinct language types (imperative languages vs. purely functional languages)

Another approach: explicitly separate effect-ful and effect-free parts of a language using types

- Introduce a **modality** called a **monad** in which all effect-ful computation takes place. Packaged effect-ful expressions are of **monadic type**.

11

## A Modal Framework

Start with a modal language of effects, called  $\mathcal{L}\{\text{cmd}\}$ , whose syntax is given by the following grammar:

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{cmd}(\tau)$	$\tau \text{ cmd}$
Expr	$e$	$::= x$   $\text{cmd}(m)$	$x$ $\text{cmd}(m)$
Comm	$m$	$::= \text{return}(e)$   $\text{letcmd}(e; x.m)$	$\text{return}(e)$ $\text{letcmd}(x)\text{be } e \text{ in } m$

Distinguishing two **modes**: pure (effect-free) **expressions** and impure (effect-capable) **commands**.

12

## Static Semantics for Monads

Two forms of typing judgements:

- $e : \tau$  – meaning (pure) expression  $e$  has type  $\tau$ ;
- $m \sim \tau$  – meaning command (impure expression)  $m$  only yields values of type  $\tau$ ;

13

### Typing Rules for $\mathcal{L}\{\text{cmd}\}$

$$\frac{\Gamma \vdash m \sim \tau}{\Gamma \vdash \text{cmd}(m) : \text{cmd}(\tau)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{return}(e) \sim \tau}$$

$$\frac{\Gamma \vdash e : \text{cmd}(\tau) \quad \Gamma, x : \tau \vdash m \sim \tau'}{\Gamma \vdash \text{letcmd}(e; x.m) \sim \tau'}$$

14

### Dynamic Semantics for $\mathcal{L}\{\text{cmd}\}$

$$\frac{\text{cmd}(m) \text{ val}}{\text{return}(e) \mapsto \text{return}(e')}$$

$$\frac{e \mapsto e'}{\text{return}(e) \mapsto \text{return}(e')}$$

$$\frac{e \text{ val}}{\text{return}(e) \text{ final}}$$

$$\frac{e \mapsto e'}{\text{letcmd}(e; x.m) \mapsto \text{letcmd}(e'; x.m)}$$

$$\frac{m_1 \mapsto m'_1}{\text{letcmd}(\text{cmd}(m_1); x.m_2) \mapsto \text{letcmd}(\text{cmd}(m'_1); x.m_2)}$$

$$\frac{\text{return}(e) \text{ final}}{\text{letcmd}(\text{cmd}(\text{return}(e)); x.m) \mapsto [e/x]m}$$

15

## Imperative Programming

The `let cmd` (bind) construct imposes a sequential execution order on commands. This suggests introducing some additional (shorthand) syntax for **sequential composition**.

For convenience and conciseness, first a binary **do** construct:

- **Concrete syntax:** `do{x ← m1; m2};`

Defined to be the impure expression:

- `let cmd(x)be cmd(m1) in m2`

16

## Further Additional Syntax

To allow for sequential composition of impure computations:

- **Concrete syntax:** `do{x1 ← m1, ..., xk ← mk; return(e)};`

Defined to be the iteration of the binary `do` as follows:

- `do{x1 ← m1; ... do{xk ← mk; return(e)} ...}`

17

## Integrating Effects

Monads provide a way to segregate effects, but make it impossible to have an effect occur within a pure expression, even if it is “benign”.

On the positive side, distinction between types  $\text{unit} \rightarrow \text{unit}$  and  $\text{unit} \rightarrow \text{unit}_{\text{cmd}}$  distinguishes the (pure) identity function from (impure) procedures.

On the other hand, it imposes strict ordering in the presence of **any** possible effects and, for example, requires that if an exception can arise **somewhere** in a program the entire program must be treated as impure.

18

## Modal Exceptions

Start with a modal language of effects, called  $\mathcal{L}\{\text{comm exc}\}$ , which extends  $\mathcal{L}\{\text{cmd}\}$  with syntax given by the following grammar:

Category	Item	Abstract	Concrete
Comm	$m$	$::= \text{raise}[\tau](e)$   $\text{letcomp}(e; x.m_1; y.m_2)$	$\text{raise}(e)$ $\text{let cmd}(x)\text{be } e \text{ in } m_1 \text{ow}(y) \text{ in } m_2$

The  $\text{letcmd}$  of  $\mathcal{L}\{\text{cmd}\}$  is treated as shorthand for :  
 $\text{let cmd}(x)\text{be } e \text{ in } m_1 \text{ow}(y) \text{ in raise}(y)$

19

## Typing Rules for $\mathcal{L}\{\text{comm exc}\}$

$$\frac{\Gamma \vdash e : \tau_{\text{exn}}}{\Gamma \vdash \text{raise}[\tau](e) \sim \tau}$$

$$\frac{\Gamma \vdash e : \text{cmd}(\tau) \quad \Gamma, x : \tau \vdash m_1 \sim \tau' \quad \Gamma, y : \tau_{\text{exn}} \vdash m_2 \sim \tau'}{\Gamma \vdash \text{letcomp}(e; x.m_1; y.m_2) \sim \tau'}$$

20

## Dynamic Semantics for $\mathcal{L}\{\text{comm exc}\}$

$$\frac{e \mapsto e'}{\text{return}(e) \mapsto \text{return}(e')} \quad \frac{e \mapsto e'}{\text{raise}[\tau](e) \mapsto \text{raise}[\tau](e')}$$

$$\frac{e \mapsto e'}{\text{letcomp}(e; x.m_1; y.m_2) \mapsto \text{letcomp}(e'; x.m_1; y.m_2)}$$

$$\frac{m \mapsto m'}{\text{letcomp}(\text{cmd}(m); x.m_1; y.m_2) \mapsto \text{letcomp}(\text{cmd}(m'); x.m_1; y.m_2)}$$

$$\frac{e \text{ val}}{\text{letcomp}(\text{cmd}(\text{return}(e)); x.m_1; y.m_2) \mapsto [e/x]m_1}$$

$$\frac{e \text{ val}}{\text{letcomp}(\text{cmd}(\text{raise}[\tau](e)); x.m_1; y.m_2) \mapsto [e/y]m_2}$$

21

## Programming With $\mathcal{L}\{\text{comm exc}\}$

If a function of type  $\text{nat} \rightarrow \text{nat}$  might raise an exception, it must now be given the weaker type  $\text{nat} \rightarrow \text{nat}_{\text{cmd}}$ .

Makes explicit the possibility of an exception outcome.

But also makes composition of two such functions impossible, due to type mismatch, so explicit sequencing (using `do` command) is required.

22

## Monadic Store Effects

References introduce store effects to a language.

Integrating references directly into the language weakens the type system, since type  $\text{nat} \rightarrow \text{nat}$  now includes the possibility of arbitrary **side effects**. This is especially problematic for call-by-name semantics.

A **monadic** approach to store effects confines potential side effects to the command level of  $\mathcal{L}\{\text{cmd}\}$ , leaving the expression level pure.

Types now explicitly indicate possible store effects, but this causes complications in the case of “benign” effects (e.g., profiling).

23

### Monadic Store Effects

The language of monadic store effects,  $\mathcal{L}\{\text{cmd ref}\}$ , integrates mutable references into  $\mathcal{L}\{\text{cmd}\}$  by adding the constructs from  $\mathcal{L}\{\text{ref}\}$ :

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{ref}(\tau)$	$\tau \text{ ref}$
Expr	$e$	$::= l$	$l$
Comm	$m$	$::= \text{ref}(e)$   $\text{get}(e)$   $\text{set}(e_1; e_2)$	$\text{ref}(e)$ $!e$ $e_1 \leftarrow e_2$

24

### Typing Rules for $\mathcal{L}\{\text{cmd ref}\}$

$$\frac{}{\Lambda, l : \tau \vdash l : \text{ref}(\tau)}$$

$$\frac{\Lambda \Gamma \vdash e : \tau}{\Lambda \Gamma \vdash \text{ref}(e) \sim \text{ref}(\tau)}$$

$$\frac{\Lambda \Gamma \vdash e : \text{ref}(\tau)}{\Lambda \Gamma \vdash \text{get}(e) \sim \tau}$$

$$\frac{\Lambda \Gamma \vdash e_1 : \text{ref}(\tau) \quad \Lambda \Gamma \vdash e_2 : \tau}{\Lambda \Gamma \vdash \text{set}(e_1; e_2) \sim \tau}$$

25

### Dynamic Semantics for Monads

Dynamic semantics for monads structured in two parts:

- $e \mapsto e'$  for (pure) expressions.
- $m @ \mu \mapsto m' @ \mu'$  for commands (impure expressions).

Both are defined as they were for the  $\mathcal{L}$  machine and the abstract machine for references, respectively, extended with the following rules for monadic constructs.

26

### Dynamic Semantics for Monads

$$\frac{}{\text{cmd}(m) \text{ val}}$$

$$\frac{e \mapsto e'}{\text{return}(e) @ \mu \mapsto \text{return}(e') @ \mu}$$

$$\frac{e \mapsto e'}{\text{letcmd}(e; x.m) @ \mu \mapsto \text{letcmd}(e'; x.m) @ \mu}$$

$$\frac{m_1 @ \mu \mapsto m'_1 @ \mu'}{\text{letcmd}(\text{cmd}(m_1); x.m_2) @ \mu \mapsto \text{letcmd}(\text{cmd}(m'_1); x.m_2) @ \mu'}$$

$$\frac{e \text{ val}}{\text{letcmd}(\text{cmd}(\text{return}(e)); x.m) @ \mu \mapsto [e/x]m @ \mu}$$

27

### Dynamic Semantics for Monads

$$\frac{}{l \text{ val}} \quad \frac{e \mapsto e'}{\text{ref}(e) @ \mu \mapsto \text{ref}(e') @ \mu} \quad \frac{e \text{ val}}{\text{ref}(e) @ \mu \mapsto \text{return}(l) @ \mu \otimes \langle l : e \rangle}$$

$$\frac{e \mapsto e'}{\text{get}(e) @ \mu \mapsto \text{get}(e') @ \mu} \quad \frac{e \text{ val}}{\text{get}(l) @ \mu \otimes \langle l : e \rangle \mapsto \text{return}(e) @ \mu \otimes \langle l : e \rangle}$$

$$\frac{e_1 \mapsto e'_1}{\text{set}(e_1; e_2) @ \mu \mapsto \text{set}(e'_1; e_2) @ \mu}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{\text{set}(e_1; e_2) @ \mu \mapsto \text{set}(e_1; e'_2) @ \mu}$$

$$\frac{e \text{ val}}{\text{set}(l; e) @ \mu \otimes \langle l : e' \rangle \mapsto \text{return}(e) @ \mu \otimes \langle l : e \rangle}$$

28

### A Comonadic Framework

Monads are good for **global/persistent** effects like storage. Comonads better for **ephemeral/local** effects like exceptions.

A framework for comonads, called  $\mathcal{L}\{\text{comon}\}$ , is given by the following grammar:

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{box}[\chi](\tau)$	$\square_\chi \tau$
Const	$\chi$	$::= \text{tt}$   $\text{and}(\chi_1; \chi_2)$	$\top$ $\chi_1 \wedge \chi_2$
Expr	$e$	$::= \text{box}(e)$   $\text{unbox}(e)$	$\text{box}(e)$ $\text{unbox}(e)$

29

## A Comonadic Framework

Central concept is the **constrained typing judgement**  $e : \tau[\chi]$ , which states that expression  $e$  has type  $\tau$  provided the context of its evaluation satisfies constraint  $\chi$ .

Constraints vary from one situation to another, but include at least trivially the true constraint ( $\top$ ) and conjunction ( $\chi_1 \wedge \chi_2$ ).

A type of the form  $\Box_{\chi}\tau$  is called a **comonad**. It represents the type of unevaluated expressions that impose constraint  $\chi$  on their context of execution.

30

## Typing Rules for $\mathcal{L}\{\text{comon}\}$

Judgement  $\chi \text{ true}$  expresses that constraint  $\chi$  is satisfied. Minimal definition of this judgement is given by the following rules:

$$\frac{}{\top \text{ true}}$$

$$\frac{\chi_1 \text{ true} \quad \chi_2 \text{ true}}{\text{and}(\chi_1; \chi_2) \text{ true}}$$

$$\frac{\text{and}(\chi_1; \chi_2) \text{ true}}{\chi_1 \text{ true}}$$

$$\frac{\text{and}(\chi_1; \chi_2) \text{ true}}{\chi_2 \text{ true}}$$

31

## Typing Rules for $\mathcal{L}\{\text{comon}\}$

Hypothetical judgements:  $\chi_1 \text{ true}, \dots, \chi_n \text{ true} \vdash \chi \text{ true}$ .

Typing judgements:  $x_1 : \tau_1[\chi_1], \dots, x_n : \tau_n[\chi_n] \vdash e : \tau[\chi]$ , abbreviated as usual as  $\Gamma \vdash e : \tau[\chi]$

$$\frac{\chi' \vdash \chi}{\Gamma, x : \tau[\chi] \vdash x : \tau[\chi']}$$

$$\frac{\Gamma \vdash e : \tau[\chi]}{\Gamma \vdash \text{box}(e) : \Box_{\chi}\tau[\chi']}$$

$$\frac{\Gamma \vdash e : \Box_{\chi}\tau[\chi] \quad \chi' \vdash \chi}{\Gamma \vdash \text{unbox}(e) : \tau[\chi']}$$

32

## Typing Rules for $\mathcal{L}\{\text{comon}\}$

Boxed computation has comonadic type under arbitrary constraint, since it is a value that imposes no constraint on its context.

Boxed computation may be activated provided ambient constraint,  $\chi'$ , is at least as strong as the constraint of the boxed computation.

### Lemma 7 (Constraint Strengthening)

If  $\Gamma \vdash e : \tau[\chi]$  and  $\chi' \vdash \chi$ , then  $e : \tau[\chi']$

33

## Dynamic Semantics for $\mathcal{L}\{\text{comon}\}$

At this level of abstract, dynamic semantics is trivial:

$$\frac{\text{box}(e) \text{ val}}{\text{unbox}(e) \mapsto \text{unbox}(e')}$$

$$\frac{e \mapsto e'}{\text{unbox}(e) \mapsto \text{unbox}(e')}$$

$$\frac{}{\text{unbox}(\text{box}(e)) \mapsto e}$$

34

## Applying Semantics for $\mathcal{L}\{\text{comon}\}$

Consider how  $\mathcal{L}\{\text{comon}\}$  might be extended with function types:

$$\frac{\Gamma, x : \tau_1[\text{tt}] \vdash e_2 : \tau_2[\text{tt}]}{\Gamma \vdash \text{lam}[\tau_1](x.e_2) : \text{arr}(\tau_1; \tau_2)[\chi]}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau[\chi] \quad e_2 : \tau_2[\chi]}{\Gamma \vdash \text{ap}(e_1; e_2) : \tau[\chi]}$$

Let  $\text{unbox\_app}(e_1; e_2)$  be an abbreviation for  $\text{unbox}(\text{ap}(e_1; e_2))$ . Derived static semantics for this construct:

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \Box_{\chi}\tau[\chi'] \quad e_2 : \tau_2[\chi'] \quad \chi' \vdash \chi}{\Gamma \vdash \text{unbox\_app}(e_1; e_2) : \tau[\chi']}$$

35

### Comonadic Exceptions

Extend  $\mathcal{L}\{\text{comon}\}$  as follows:

Category	Item	Abstract	Concrete
Const	$\chi$	$::= \uparrow$	$\uparrow$
Expr	$e$	$::= \text{raise}[\tau](e)$   $\text{handle}(e_1; x.e_2)$	$\text{raise}(e)$ $\text{try } e_1 \text{ ow } x \Rightarrow e_2$

Constraint  $\uparrow$  specifies that an expression may raise an exception and hence that its context must provide a handler.

36

### Comonadic Exceptions

Extend static semantics of  $\mathcal{L}\{\text{comon}\}$  with the following rules:

$$\frac{\Gamma \vdash e : \tau_{\text{exn}} \quad \chi \uparrow}{\Gamma \vdash \text{raise}[\tau](e) : \tau[\chi]}$$

$$\frac{\Gamma \vdash e_1 : \tau[\chi \wedge \uparrow] \quad \Gamma, x : \tau_{\text{exn}} \vdash e_2 : \tau[\chi]}{\Gamma \vdash \text{handle}(e_1; x.e_2) : \tau[\chi]}$$

Dynamic semantics is unchanged, but interesting issue is how comonadic typing provides additional assurances. This is formalized by viewing control stack as a constraint transformer.

37

### Stack Typing

A stack represents the work remaining to complete a computation, so  $k : \tau$  means stack  $k$  transforms a value of type  $\tau$  into a value of type  $\tau'$

This judgement is inductively defined by:

$$\frac{\overline{e : \tau[\uparrow]}}{k : \tau'[\chi'] \quad f : \tau[\chi] \Rightarrow \tau'[\chi']}{k; f : \tau[\chi]}$$

38

### Stack Typing

An example stack frame typing rule:

$$\frac{x : \tau_{\text{exn}} \vdash e : \tau[\chi]}{\text{handle}(-; x.e) : \tau[\chi \wedge \uparrow] \Rightarrow \tau[\chi]}$$

39

### Well-Formed States for the $\mathcal{K}\{\text{nat} \rightarrow\}$ Abstract Machine

The two forms of state for the  $\mathcal{K}\{\text{nat} \rightarrow\}$  Machine are well-formed provided that their stack and expression components match.

This judgement is inductively defined by:

$$\frac{k : \tau[\chi] \quad e : \tau[\chi]}{k \triangleright e \text{ ok}}$$

$$\frac{k : \tau[\chi] \quad e : \tau[\chi] \quad e \text{ val}}{k \triangleleft e \text{ ok}}$$

Safety ensures that no uncaught exceptions can arise. Formalized by defining final states to be only those returning a value to an empty stack.

$$\frac{e \text{ val}}{e \triangleleft e \text{ final}}$$

40

### Type Safety

#### Theorem 8 (Preservation)

If  $s \text{ ok}$  and  $s \mapsto s'$ , then  $s' \text{ ok}$ .

#### Theorem 9 (Progress)

If  $s \text{ ok}$  then either  $s$  final, or there exists  $s'$  such that  $s \mapsto s'$ .

No uncaught exception case in this theorem!

41