Dynamic Classification

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( r )</td>
<td>( \text{clsfd} )</td>
<td>( \text{clsfd} )</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{in}(a)(e) )</td>
<td>( \text{in}(a)(e) )</td>
</tr>
<tr>
<td>Rule</td>
<td>( r )</td>
<td>( \text{in}\Gamma(a)(x,e) )</td>
<td>( \text{in}(a)(x) \to e )</td>
</tr>
</tbody>
</table>

Dynamic Semantics for \( \mathcal{L} \) (classified)

Using same abstract machine as for \( \mathcal{L} \) (fluid new):

\[
\begin{align*}
e \vdash e & \quad \text{val} \\
\text{in}(a)(e) & \quad \text{val} \\
\text{e} \uplus \nu & \Rightarrow e \uplus \nu' \\
\text{in}(a)(e) & \Rightarrow \text{in}(a)(e) \uplus \nu \\
\text{ccase}(e; r_0, \ldots, r_n) & \Rightarrow \text{ccase}(e; r_0, \ldots, r_n) \\
\text{ccase}(e; c, r_0, \ldots, r_n) & \Rightarrow \text{ccase}(e; c, r_0, \ldots, r_n) \\
\end{align*}
\]

Dynamic Semantics for \( \mathcal{L} \) (classified)

Theorem 1 (Preservation)

Suppose that \( e \uplus \nu \Rightarrow e' \uplus \nu' \), where \( \Gamma \vdash e : \tau \) and \( \Gamma \vdash a : \tau \)

whenever \( a \in \nu \). Then \( \nu' \supseteq \nu \), and there exists \( \Sigma' \supseteq \Sigma \) such that

\( \Sigma' \vdash e' : \tau \) and \( \Sigma' \vdash a' : \tau \) for each \( a' \in \nu' \).

Proof is by induction on evaluation.
The canonical forms lemma characterizes closed values of $\mathcal{L}_{\text{classified}}$:

**Lemma 2**
Suppose that $\Sigma \vdash e : \text{class}$ and $e \text{ val}$. Then $e = \text{in}(a)(e')$ for some $a$ such that $\Sigma \vdash a : \tau$ and some $e'$ such that $e' \text{ val}$ and $\Sigma \vdash e' : \tau$.

**Theorem 3 (Progress)**
Suppose that $\Sigma \vdash e : \tau$ and that if $a \in \nu$, then $\Sigma \vdash a : \tau_0$ for some type $\tau_0$. Then either $e \text{ val}$ or $e \nu \rightarrow e' \text{ val}$ for some $e'$ and $\nu'$.

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$::= \text{class}(\tau)$</td>
<td>$\tau_{\text{class}}$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$::= \text{clas}(a)$</td>
<td>$\text{clas}(a)$</td>
</tr>
<tr>
<td></td>
<td>$e$</td>
<td>$::= \text{case}(r{:}\cdot;\rho;\rho_1;\ldots;\rho_n)$</td>
<td>$\text{case}(r{:}\cdot;\rho;\rho_1;\ldots;\rho_n)$</td>
</tr>
<tr>
<td>Rule</td>
<td>$r$</td>
<td>$::= \text{clas}(a)(e)$</td>
<td>$\text{clas}(a) \rightarrow e$</td>
</tr>
</tbody>
</table>

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

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

**Lemma 5**
Suppose that $\Sigma \vdash e : \tau \cdot \text{class}$ and $e \text{ val}$. Then $e = \text{clas}(a)$ 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 $\Sigma \vdash a : \tau_0$ for some type $\tau_0$. Then either $e \text{ val}$ or $e \nu \rightarrow e' \text{ val}$ for some $e'$ and $\nu'$.

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.

### 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 $\mathcal{L}_{\text{fluid}}$ using the following syntax.

### 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 $\mathcal{L}_{\text{fluid}}$.

**Theorem 4 (Preservation)**
Suppose that $e \nu \rightarrow e' \nu'$, where $\Sigma \vdash e : \tau$ and $\Sigma \vdash a : \tau_0$ whenever $a \in \nu$. Then $\nu' \supseteq \nu$, and there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash e' : \tau$ and $\Sigma' \vdash a' : \tau_0$ for each $a' \in \nu'$.

Proof is by induction on evaluation.

### 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 effectful and effect-free parts of a language using types

- Introduce a modality called a **monad** in which all effectful computation takes place. Packaged effectful expressions are of monadic type.
A Modal Framework

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\tau$</td>
<td>$cmd(\tau)$</td>
<td>$\tau cmd$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$x$</td>
<td>$cmd(m)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$cmd(m)$</td>
<td>$cmd(m)$</td>
</tr>
<tr>
<td>Comm</td>
<td>$m$</td>
<td>$return(e)$</td>
<td>$return(e)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$let cmd(x; m)$</td>
<td>$let cmd(x; m)$</td>
</tr>
</tbody>
</table>

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

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 $\tau$;

Typing Rules for $L_{cmd}$

<table>
<thead>
<tr>
<th>$\Gamma \vdash m \sim \tau$</th>
<th>$\Gamma \vdash cmd(n) \sim cmd(\tau)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash e : \tau$</td>
<td>$\Gamma \vdash x : \tau$</td>
</tr>
<tr>
<td>$\Gamma \vdash cmd(r)$</td>
<td>$\Gamma \vdash let cmd(x; m) \sim \tau'$</td>
</tr>
</tbody>
</table>

Dynamic Semantics for $L_{cmd}$

<table>
<thead>
<tr>
<th>$cmd(m) \ val$</th>
<th>$e \mapsto e'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$return(e) \ sim return(e')$</td>
<td></td>
</tr>
<tr>
<td>$e \ val$</td>
<td></td>
</tr>
<tr>
<td>$return(e) \ final$</td>
<td></td>
</tr>
<tr>
<td>$let cmd(c; x; m) \ sim let cmd(c'; x; m)$</td>
<td></td>
</tr>
<tr>
<td>$m_1 \ sim m'_1$</td>
<td></td>
</tr>
<tr>
<td>$let cmd(cmd(m'_1); x; m_2) \ sim let cmd(c; x; m)$</td>
<td></td>
</tr>
<tr>
<td>$let cmd(cmd(c); x; m) \ sim [e/x]m$</td>
<td></td>
</tr>
</tbody>
</table>

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:** $do\{x \leftarrow m_1; m_2\}$;

Defined to be the impure expression:

- $let cmd(x) be cmd(m_1) in m_2$

Further Additional Syntax

To allow for sequential composition of impure computations:

- **Concrete syntax:** $do\{x_1 \leftarrow m_1; \ldots; x_k \leftarrow m_k; return(e)\}$;

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

- $do\{x_1 \leftarrow m_1; \ldots; x_k \leftarrow m_k; return(e)\} \ldots$
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 unit → unit and unit → unit 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.

**Typing Rules for L[com exc]**

<table>
<thead>
<tr>
<th>Context</th>
<th>Expression</th>
<th>Type Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ e :: τ if</td>
<td>return(r)</td>
<td>Γ ⊢ return(r)</td>
</tr>
<tr>
<td>Γ ⊢ e :: τ if</td>
<td>raise(τ)[e]</td>
<td>Γ ⊢ raise(τ)[e]</td>
</tr>
<tr>
<td>Γ ⊢ e :: τ if</td>
<td>letcomp(x; m₁, y; m₂)</td>
<td>Γ ⊢ letcomp(x; m₁, y; m₂)</td>
</tr>
</tbody>
</table>

**Dynamic Semantics for L[com exc]**

<table>
<thead>
<tr>
<th>Context</th>
<th>Expression</th>
<th>Value Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>e = e’ if</td>
<td>return(r)</td>
<td>e = return(r)</td>
</tr>
<tr>
<td>e = e’ if</td>
<td>raise(τ)[e]</td>
<td>e = raise(τ)[e]</td>
</tr>
<tr>
<td>m = m’ if</td>
<td>letcomp(x; m₁, y; m₂)</td>
<td>m = letcomp(x; m₁, y; m₂)</td>
</tr>
</tbody>
</table>

**Programming With L[com exc]**

If a function of type sat → sat might raise an exception, it must now be given the weaker type sat → sat cmd.

Makes explicit the possibility of an exception outcome.

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

**Modal Exceptions**

Start with a modal language of effects, called L[com exc], which extends L[cmd] with syntax given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm</td>
<td>m</td>
<td>raise(e)</td>
<td>letcmd(x; m₁, y; m₂)</td>
</tr>
</tbody>
</table>

The letcmd of L[cmd] is treated as shorthand for:

letcmd(x; m₁, y; m₂) in raise(y)

**Monad**

References introduce store effects to a language.

Integrating references directly into the language weakens the type system, since type sat → sat 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 L[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).
Monadic Store Effects

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{ref} { \tau } )</td>
<td>( \tau ) ref</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{ref} { e } )</td>
<td>( e )</td>
</tr>
<tr>
<td>Comm</td>
<td>( m )</td>
<td>( \text{ref} { e } )</td>
<td>( m )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{ref} { e } )</th>
<th>( \text{ref} { e } )</th>
<th>( \text{ref} { e } )</th>
<th>( \text{ref} { e } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{get} )</td>
<td>( \text{get} )</td>
<td>( \text{get} )</td>
<td>( \text{get} )</td>
</tr>
<tr>
<td>( \text{set} { e_1, e_2 } )</td>
<td>( e_1 )</td>
<td>( e_2 )</td>
<td>( e_1 )</td>
</tr>
</tbody>
</table>

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 \( L \) machine and the abstract machine for references, respectively, extended with the following rules for monadic constructs.

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 \( L \{ \text{comaa} \} \), is given by the following grammar:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{box} { \chi } { \tau } )</td>
<td>( \chi ) ( \tau )</td>
</tr>
<tr>
<td>Const</td>
<td>( \chi )</td>
<td>( \text{tt} )</td>
<td>( \top )</td>
</tr>
<tr>
<td></td>
<td>( \chi )</td>
<td>( \text{and} { \chi_1 \times \chi_2 } )</td>
<td>( \chi_1 \times \chi_2 )</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{box} { e } )</td>
<td>( \text{box} { e } )</td>
</tr>
<tr>
<td></td>
<td>( e )</td>
<td>( \text{unbox} { e } )</td>
<td>( \text{unbox} { e } )</td>
</tr>
</tbody>
</table>
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 $(\chi_1 \land \chi_2)$.

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

Typing Rules for $\mathcal{L}[\text{conon}]$

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

Typing judgements: $\Gamma : \chi_1 \tau_1, \ldots, \tau_n \chi_n \vdash e : \tau(\chi)$, abbreviated as usual $\Gamma \vdash e : \tau(\chi)$

\[
\begin{align*}
\Gamma, x : \tau(\chi) \vdash x : \tau(\chi) \\
\Gamma \vdash e : \tau(\chi) \\
\Gamma \vdash \chi_1 \Box e : \Omega(\chi_1) \\
\Gamma \vdash e : \Box \chi(\chi') \chi' \vdash \chi \\
\Gamma \vdash \text{unbox}(e) : \tau(\chi)
\end{align*}
\]

Typing Rules for $\mathcal{L}[\text{conon}]$

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 $\Gamma \vdash e : \tau(\chi')$.

Dynamic Semantics for $\mathcal{L}[\text{conon}]$

At this level of abstract, dynamic semantics is trivial:

\[
\begin{align*}
\text{Box}(c) \text{ val} & \quad e \mapsto e' \\
\text{unbox}(e) & \mapsto \text{unbox}(e') \\
\text{unbox(box}(e)) & \mapsto e
\end{align*}
\]

Applying Semantics for $\mathcal{L}[\text{conon}]$

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

\[
\Gamma \vdash e_1 : \chi_1 \tau_1, \ldots, e_n : \chi_n \tau_n \\
\Gamma \vdash \text{ap}(e_1, e_2) : \tau(\chi_1)
\]

Let $\text{unbox}_{\text{ap}}(e_1, e_2)$ be an abbreviation for $\text{unbox}(\text{ap}(e_1, e_2))$.

Derived static semantics for this construct:

\[
\begin{align*}
\Gamma \vdash e_1 : \Box \chi(\chi') \chi' \vdash \chi \\
\Gamma \vdash \text{unbox}_{\text{ap}}(e_1, e_2) : \tau(\chi')
\end{align*}
\]
### Comonadic Exceptions

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>$\chi$</td>
<td>$\n \n \chi$</td>
<td>$\n \n \chi$</td>
</tr>
<tr>
<td>Expr</td>
<td>$e$</td>
<td>$\n \n \text{raise}(\tau(e))$</td>
<td>$\n \n \text{raise}(\tau(e))$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\n \n \text{handle}(e; x; e2)$</td>
<td>$\n \n \text{try}(e \circ e2)$</td>
</tr>
</tbody>
</table>

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

### 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:

$$
\frac{k : \tau(x) \quad f : \tau(x) \Rightarrow \tau'(x)}{k, f : \tau'(x)}
$$

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

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

This judgement is inductively defined by:

$$
\frac{k : \tau(x) \quad e : \tau(x)}{k, e : \tau(x) \quad e \ \text{val}}
\quad
\frac{e \ \text{val}}{e \circ e \ \text{final}}
$$

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

### Comonadic Exceptions

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

$$
\frac{\Gamma \vdash e : \text{com} \quad \chi \vdash \top}{\Gamma \vdash \text{raise}(\tau(e)) : \tau(x)}
$$
$$
\frac{\Gamma, x : \text{com} \vdash \nu : \tau(x)}{\Gamma \vdash \text{handle}(e_1; x; e_2) : \tau(x)}
$$

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.

### Type Safety

**Theorem 8 (Preservation)**

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

**Theorem 9 (Progress)**

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

No uncaught exception case in this theorem!