A logical interpretation of Java-style exceptions

Jeff Vaughan

Department of Computer Science
Harvard University

CL&C, August 22, 2010
Pop Quiz!

Ignore divergence and mutable state. What is the logical content of the following program?

```java
D m(C arg) throws E F {
  ... 
}
```
Pop Quiz!

Ignore divergence and mutable state. What is the logical content of the following program?

D m(C arg) throws E F { ... }

Answer: C ⊨ D ∨ E ∨ F
There are many logic-based readings of exceptions.

[Nakano ’92, ’94] Introduced exceptions in an extension of system LJ’.


[Kameyama ’97] Exceptions in Gödel’s T.

[De Groote ’95] Exceptions are named by lexically-scoped variables, with classical typing rules.

[Ong & Steward ’97] Exceptions names are covariables in $\mu$PFC.
Beautiful models, but far from practical languages.

- For example, Nakano ’92:
  - Exception names are represented by lexically-scoped *tags*.
  - Many administrative tag abstractions and instantiations.
  - Latent effects must be manually suspended as part of function definitions.

- *Type-and-effect* analyses address these problems, but have received little attention from a logic perspective.

  [Lucassen & Gifford ’88, Talpin & Jouvelot ’92]
This talk: Finding the logical-content of exceptions, from the perspective of type-and-effect analysis.

1. System EC: An exception calculus
2. Embedding of EC in classical logic
3. Future directions
System EC: An exception calculus
EC models Java-style exceptions.

- Exceptions are first class values, and are identified by type name.
- Checked exception methodology requires that functions be annotated with a set of throwable exceptions.
- Subtyping lets one exception handler catch multiple related exceptions.
- Call-by-value semantics enable precise reasoning.

Focusing on exceptions: no classes, divergence, or state.
EC’s expression language extends lambda calculus.

### Definition (Expression Syntax)

\[
e ::= x | e_1 e_2 | \lambda x : \tau. e \quad \text{Lambda calculus}
\]

- \text{top} \quad \text{Top/unit value}
- \text{E e} \quad \text{Exception expression}
- \text{raise e} \quad \text{Throw exception}
- \text{e_1 handle E x \Rightarrow e_2} \quad \text{Exception handler}

Example (Evaluation)

```
\lambda x : \top. \text{raise Fail} \\
\text{ohno!}
```
EC’s expression language extends lambda calculus.

Definition (Expression Syntax)

\[ e ::= \ x \mid e_1 \ e_2 \mid \lambda x: \tau. \ e \]

- Lambda calculus
- Top/unit value
- Exception expression
- Throw exception
- Exception handler

Example (Evaluation)

\[(\lambda x: \text{top}. \text{raise } \text{Fail "ohno!"} ) \ 1 \ \text{handle} \ \text{Any} \ x \Rightarrow 2\]

\[\text{\rightarrow}^*\]

\[2\]
Type-and-effects–style analysis tracks exceptions.

\[ e \equiv \text{if } b \text{ then } 3 \text{ else (raise } \text{Fail "oops"}) \]

\[ e: \quad \text{ret: int} \quad \text{exn Fail: string} \]
Type-and-effects–style analysis tracks exceptions.

\[ e \equiv \text{if } b \text{ then } 3 \text{ else } (\text{raise } \text{Fail } "\text{oops"}) \]

\[ e: \quad \text{ret}: \text{int} , \quad \text{exn Fail}: \text{string} \]

Type of normal termination
Type-and-effects–style analysis tracks exceptions.

\[ e \equiv \text{if } b \text{ then } 3 \text{ else (raise } \text{Fail "oops"}) \]

\[ e: \quad \text{ret: int, exn Fail: string} \]

Type of normal termination

List of possible exceptions
Type-and-effects–style analysis tracks exceptions.

\[ e \equiv \textbf{if } b \textbf{ then } 3 \textbf{ else } (\text{raise } \text{Fail } "\text{oops}" ) \]

\[ e : \quad \textbf{ret} : \text{int} , \quad \textbf{exn} \text{ Fail} : \text{string} \]

\[ \lambda b : \text{bool} . \; e : \quad \textbf{ret} : (\text{bool } \rightarrow (\textbf{ret} : \text{int} , \quad \textbf{exn} \text{ Fail} : \text{string} )) \]
Type-and-effects–style analysis tracks exceptions.

\[ e \equiv \text{if } b \text{ then } 3 \text{ else } (\text{raise } \text{Fail} \ "\text{oops}\") \]

\[ e : \quad \text{ret} : \text{int} , \quad \text{exn Fail} : \text{string} \]

\[ \lambda b : \text{bool}. \ e : \quad \text{ret} : (\text{bool} \rightarrow (\text{ret} : \text{int}, \ \text{exn Fail} : \text{string})) \]

Values always return
Type-and-effects–style analysis tracks exceptions.

\[ e \equiv \text{if } b \text{ then } 3 \text{ else (raise Fail "oops")} \]

\[ e : \ ret : \text{int} , \ \text{exn Fail : string} \]

\[ \lambda \ b : \text{bool}. \ e : \ \text{ret : (bool } \rightarrow (\text{ret : int} , \text{exn Fail : string } )) \]

Values always return

Function body exceptions are captured as *latent effects*
Exception typing uses signatures and result contexts.

### Form of typing judgment

\[ e: \Sigma; \Gamma \vdash \Delta \]

### Definition

<table>
<thead>
<tr>
<th>( \Sigma ) ::=</th>
<th>Empty Signature</th>
<th>Exception declaration</th>
<th>Subtype declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \cdot )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Sigma, E \sim \tau )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Sigma, E_1 &lt;: E_2 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \Delta ) ::=</th>
<th>Result Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \cdot )</td>
<td></td>
</tr>
<tr>
<td>( \Delta, \text{ret} : \tau )</td>
<td></td>
</tr>
<tr>
<td>( \Delta, \text{exn} : \tau )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \tau ) ::=</th>
<th>Base types and top</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>Arrow</td>
</tr>
<tr>
<td>( \tau \rightarrow \Delta )</td>
<td></td>
</tr>
<tr>
<td>( \text{Exn} E )</td>
<td>Exception type</td>
</tr>
</tbody>
</table>
App & abs rules are specialized for latent effects

\[
\begin{align*}
e & : \Sigma; \Gamma, x : \tau \vdash \Delta \\
\lambda x : \tau. \ e & : \Sigma; \Gamma \vdash \text{ret} : \tau \to \Delta \\
\hline
\end{align*}
\]

\[T-\text{ABS}\]

\[
\begin{align*}
e_1 & : \Sigma; \Gamma \vdash \text{ret} : \tau_2 \to (\text{ret} : \tau, \Delta), \Delta \\
e_2 & : \Sigma; \Gamma \vdash \text{ret} : \tau_2, \Delta \\
\hline
\end{align*}
\]

\[T-\text{APP}\]

\[
\begin{align*}
e_1 \; e_2 & : \Sigma; \Gamma \vdash \text{ret} : \tau, \Delta \\
\end{align*}
\]

※ Computational effects are suspended at abstractions and restored at applications.
Subtyping is a useful programming language feature.

\[ \Sigma \equiv Disk <: IO, Sound <: IO, IO <: Any, \ldots \]

\[ e : \Sigma ; \vdash \text{ret: int, exn Disk: string, exn Sound: } \top \]

**Example (Polymorphic exception handling)**

\[ e \text{ handle } IO \ x \Rightarrow e' \]

catches all Disk, Sound, and IO exceptions.

**Example (Subtyping allows conservative typings)**

\[ e : \Sigma ; \vdash \text{ret: int, exn Any: } \top \]
Embedding of EC in classical logic
EC typing derivations give rise to LK proofs trees.

- Each EC type corresponds to an LK proposition.
  - Mostly standard interpretation
    - [Curry, Feys & Craig ’58, Howard ’80]
  - Latent effects are represented by disjunction.

- EC subtyping translates to logical entailment.

- EC typing derivations translate to LK derivations.
LK is the canonical classical sequent calculus.

**Definition (LK Propositions)**

\[ P, Q \ ::= \ A_{LK} \mid \top \mid \bot \mid P \land Q \mid P \lor Q \mid P \supset Q \mid \neg P \]

Judgment form

\[ P_1 \ldots P_n \vdash Q_1 \ldots Q_m \]

“means” the conjunction of the \( P \)'s implies the disjunction of the \( Q \)'s.

[Gentzen ’35]
EC types translate to LK propositions.

Definition $([[]]_\cdot)$

\[
[[ A ]]_{\text{typ}}^\Sigma = A_{LK} \\
[[ \top ]]_{\text{typ}}^\Sigma = \top \\
[[ \text{Exn } E ]]_{\text{typ}}^\Sigma = [[ \tau ]]_{\text{typ}}^\Sigma \quad \text{where } E \sim \tau \in \Sigma \\
[[ \tau \rightarrow \Delta ]]_{\text{typ}}^\Sigma = [[ \tau ]]_{\text{typ}}^\Sigma \supset \bigvee [[ \Delta ]]_{\text{ctx}}^\Sigma \\
[[ \cdot ]]_{\text{ctx}}^\Sigma = [[ \cdot ]]_{\text{env}}^\Sigma = . \\
[[ \Gamma, \tau \mapsto ]}_{\text{env}}^\Sigma = [[ \Gamma ]]_{\text{env}}^\Sigma, [[ \tau ]]_{\text{typ}}^\Sigma \\
[[ \Delta, \tau \mapsto ]}_{\text{ctx}}^\Sigma = [[ \Delta ]]_{\text{ctx}}^\Sigma, [[ \tau ]]_{\text{typ}}^\Sigma \\
\bigvee P_1, P_2, \ldots, P_n = P_1 \lor P_2 \lor \cdots \lor P_n
Main result: typing and subtyping have logical content.

Lemma (Subtyping)

Suppose $\mathcal{D}$ a subtyping derivation in EC.

- If $\mathcal{D} :: \Sigma \vdash \Delta_1 <: \Delta_2$ then $\llbracket \Delta_1 \rrbracket^\Sigma_{ctx} \mapsto \llbracket \Delta_2 \rrbracket^\Sigma_{ctx}$. 
- If $\mathcal{D} :: \Sigma \vdash \tau_1 <: \tau_2$ then $\llbracket \tau_1 \rrbracket^\Sigma_{typ} \mapsto \llbracket \tau_2 \rrbracket^\Sigma_{typ}$.
- If $\mathcal{D} :: \Sigma \vdash \Diamond$ then $\llbracket \tau_1 \rrbracket^\Sigma_{typ} \mapsto \llbracket \tau_2 \rrbracket^\Sigma_{typ}$ where $E_1 <: E_2$, $E_1 \sim \tau_1$, $E_2 \sim \tau_2 \in \Sigma$.

Theorem (Shallow embedding of EC in LK)

Suppose $e: \Sigma; \Gamma \vdash \Delta$. Then $\llbracket \Gamma \rrbracket^\Sigma_{env} \mapsto \llbracket \Delta \rrbracket^\Sigma_{ctx}$. 

17/21
Future directions
Cam we move from an embedding to an isomorphism?

- EC is (likely) constructive, but LK is classical—(likely) no way to translate LK proofs to EC typing derivations.

- LJ’ may be a good translation target.
  - LJ’ is an intuitionistic variant of LK.
  - LJ’ has multiple conclusions—these seem essential.
  - LJ’ restricts the LK implication and negation rules:

\[
\begin{align*}
\text{LK-IMP} & : \quad P, \Phi \vdash \Theta, Q \\
& \quad \Rightarrow \quad \Phi \vdash \Theta, P \supset Q \\
\text{LJ’-IMP} & : \quad P, \Phi \vdash Q \\
& \quad \Rightarrow \quad \Phi \vdash P \supset Q
\end{align*}
\]

- Implication restriction looks like a good fit with EC’s handling of latent effects.

[Takeuti ’75]
Can we logically interpret other effects systems?

- Possible nontermination $\Rightarrow$ Local absence of logical content?
  - *Termination casts* [Stump, Sjöberg, and Weirich ’10]

- World effects $\Rightarrow$ An alternative means to model context?
  - *Contextual modal type theory* [Nanevski, Pfenning, and Pientka ’08]
Conclusions

■ This talk: EC typing derivations give rise to LK proofs.
■ Generally: Effects systems can have a logical interpretation.
■ Many interesting problems remain!
Conclusions

- This talk: EC typing derivations give rise to LK proofs.
- Generally: Effects systems can have a logical interpretation.
- Many interesting problems remain!

Thank you!