CS150 APL: Effects

Guannan Wei
guannan.wei@tufts.edu

Sept 30, 2025

Tufts University

Logistics

- Thursday (Oct 2): project proposal presentation (15 min)
- Sunday (Oct 5): 1-page project proposal due
  - LaTeX template on Canvas

Last time

- Universal and existential types
- Product and sum types
- Mutable references

Today's topic

Control effects:

- Exceptions
- Algebraic effects
- Continuations

Exceptions

- `try-catch` in Java and many other languages
- An example in Java:

```java
try {
  // code that may throw an exception
  ...
  throw ex;
  ...
} catch (Exception e) {
  // handler for Exception
}
```

Exceptions

Syntax

$$
\begin{array}{rcll}
n & \in & \mathbb{N} & \\
v & ::= & n \mid \lambda x.t & \textbf{values} \\
t & ::= & n \mid x \mid \lambda x.t \mid t_1\, t_2 \mid t_1 \oplus t_2 & \textbf{terms} \\
  & \mid & \texttt{throw}\ v \mid \texttt{try}\ t_1\ \texttt{catch}\ x.\ t_2 &
\end{array}
$$

Dynamics (first attempt)

$$E ::= \square \mid v\,E \mid E\,t \mid v \oplus E \mid E \oplus t \quad \textbf{reduction contexts}$$
$$\mid \quad \texttt{try}\ E\ \texttt{catch}\ x.\ t$$

$$\frac{}{(\lambda x.t)\,v \to t[x := v]}\ \beta_v \qquad \frac{}{n_1 \oplus n_2 \to n_1 + n_2}\ \textsc{Add}$$

$$\frac{}{\texttt{try}\ v\ \texttt{catch}\ x.\ t \to v}\ \textsc{Return} \qquad \frac{}{\texttt{try}\ E[\texttt{throw}\ v]\ \texttt{catch}\ x.\ t \to t[x := v]}\ \textsc{Catch}$$

$$\frac{t_1 \to t_1'}{E[t_1] \to E[t_1']}\ \textsc{Ctx}$$

Exceptions

Example:

```
try {
  try { throw 42 } catch x. { x + 1 }
} catch y. { y + 2 }
```

Exceptions

Example:

```
try {
  try { throw 42 } catch x. { x + 1 }
} catch y. { y + 2 }
```

Problem: ambiguous decomposition of E!

Exceptions

Dynamics

$$
\begin{array}{rcll}
E & ::= & \square \mid v\,E \mid E\,t \mid v \oplus E \mid E \oplus t & \textbf{local contexts} \\
E_h & ::= & \square \mid v\,E_h \mid E_h\,t \mid v \oplus E_h \mid E_h \oplus t & \textbf{handler contexts} \\
 & \mid & \texttt{try}\ E_h\ \texttt{catch}\ x.\ t &
\end{array}
$$

$$\frac{}{(\lambda x.t)\,v \to t[x := v]}\ \beta_v \qquad \frac{}{n_1 \oplus n_2 \to n_1 + n_2}\ \textsc{Add}$$

$$\frac{}{\texttt{try}\ v\ \texttt{catch}\ x.\ t \to v}\ \textsc{Return} \qquad \frac{}{\texttt{try}\ E[\texttt{throw}\ v]\ \texttt{catch}\ x.\ t \to t[x := v]}\ \textsc{Catch}$$

$$\frac{t_1 \to t_1'}{E_h[t_1] \to E_h[t_1']}\ \textsc{Ctx}_h$$

Exceptions

Example:

```
     try { try { 10+(throw 42) } catch x. { x + 1 } } catch y. { y + 2 }
     /* Catch */
->   try { 42 + 1 } catch y. { y + 2 }
     /* Return */
->*  43
```

Exceptions

- Error recovery: what if we want to recover from an error and continue?

Resumable Exceptions

- Generalization of exceptions: `catch` also binds a "resumption" that can be invoked to resume the computation where the effect was raised.

```
try {
  val x = throw v;
  // using x
  ...
} catch x,k. {
  ...
  k(v)
}
```

Algebraic Effects

- This idea is known as **algebraic effects and handlers**; expressive and modular way to write effectful programs.
- Mainstream languages such as OCaml 5 have adopted effects handlers.

Algebraic Effects

- This idea is known as **algebraic effects and handlers**; expressive and modular way to write effectful programs.
- Mainstream languages such as OCaml 5 have adopted effects handlers.
- Demo: the Eff language

Algebraic Effects

- Expresiveness: powerful control abstraction
  - nondeterminism, backtracking
  - mutable states
  - coroutines, async/await
  - etc.
- Modularity:
  - Allow flexible user-defined effect operations
  - Handlers are defined separately
  - Composing multiple effects and handlers is easy

Algebraic Effects

- A fine-grained call-by-value lambda calculus with algebraic effects and handlers

Syntax

$$
\begin{array}{lcll}
n & \in & \mathbb{N} & \\
v & ::= & n \mid x \mid \lambda x.t & \textbf{values} \\
t & ::= & v \mid \texttt{return}\ v \mid v_1\ v_2 \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 & \textbf{computations} \\
 & \mid & \texttt{do}\ v \mid \texttt{handle}\ t\ \texttt{with}\ x.t_1; x, k.t_2 &
\end{array}
$$

Dynamics

$$F \quad ::= \quad \Box \mid \texttt{let}\ x = F\ \texttt{in}\ t$$ **pure contexts**

$$E \quad ::= \quad \Box \mid \texttt{let}\ x = E\ \texttt{in}\ t \mid \texttt{handle}\ E\ \texttt{with}\ x.t_1; x, k.t_2$$ **general contexts**

$$\frac{}{(\lambda x.t)\ v \to t[x := v]}\ \beta_v \qquad \frac{}{\texttt{let}\ x = \texttt{return}\ v\ \texttt{in}\ t \to t[x := v]}\ \textsc{Let}$$

$$\frac{}{\texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ x.t_1; x, k.t_2 \to t_1[x := v]}\ \textsc{Return}$$

$$\frac{f = \lambda y.\texttt{handle}\ F[\texttt{return}\ y]\ \texttt{with}\ x.t_1; x, k.t_2}{\texttt{handle}\ F[\texttt{do}\ v]\ \texttt{with}\ x.t_1; x, k.t_2 \to t_2[x := v, k := f]}\ \textsc{Handle}$$

Algebraic Effects

Example:

```
handle {
  let x = do 2 in
  let y = do 3 in
  return (x + y)
} with {
  x => return x
  x,k => k(x * 2)
}
```

Algebraic Effects

Example:

```
handle {
  let x = do 2 in
  let y = do 3 in
  return (x + y)
} with {
  x => return x
  x,k => k(x * 2)
}
```

```
  k(x * 2)
where
  x = 2
  k = \z. handle {
    let x = return z in
    let y = do 3 in
    return (x + y)
  } with {
    x => return x
    x,k => k(x * 2)
  }
```

Algebraic Effects

Example (cont'd):

```
handle {
  let x = return 4 in
  let y = do 3 in
  return (x + y)
} with {
  x => return x
  x,k => k(x * 2)
}
```

```
handle {
  let y = do 3 in
  return (4 + y)
} with {
  x => return x
  x,k => k(x * 2)
}
```

Algebraic Effects

Further reading

- Tutorial: *An Introduction to Algebraic Effects and Handlers. Matija Pretnar* https://www.eff-lang.org/handlers-tutorial.pdf
- Theory: Why "algebraic"? Because effects can be modeled using algebraic theories.

  What is algebraic about algebraic effects and handlers? Andrej Bauer https://arxiv.org/abs/1807.05923
- Implementation: CEK-style abstract machine for algebraic effects and handlers.

  Liberating Effects with Rows and Handlers. Hillerstrom and Lindley. TyDE '16

Algebraic Effects and Continuations

- Effect handlers captures "delimited continuations" (i.e. rest of computation up to nearest handler).
- A family of general delimited control operators:
  - `shift`/`reset` (Abstracting Control, Danvy and Filinski)
  - `control`/`prompt` (The theory and practice of first-class prompts, Felleisen)
  - `shift0`/`reset0` and `control0`/`prompt0` (Shift to Control, Shan)
- You can try them in Racket!

Delimited Continuations

A λ-calculus with `shift`/`reset`:

Syntax and dynamics

$$t \quad ::= \quad \mid x \mid \lambda x.t \mid t_1\, t_2 \mid \langle t \rangle \mid \texttt{shift}\ k.t \qquad \textbf{terms}$$

$$E \quad ::= \quad \Box \mid v\, E \mid E\, t \mid \langle E \rangle \qquad \textbf{reduction contexts}$$

$$\frac{}{(\lambda x.t)\, v \to t[x := v]}\ \beta_v \qquad \frac{}{\langle v \rangle \to v}\ \textsc{Reset}$$

$$\frac{E \text{ does not contain } \langle \cdot \rangle}{\langle E[\texttt{shift}\ k.t] \rangle \to t[k := \lambda x.\langle E[x] \rangle]}\ \textsc{Shift} \qquad \frac{t_1 \to t_1'}{E[t_1] \to E[t_1']}\ \textsc{Ctx}$$

Delimited Continuations

- Some cool applications of algebraic effects and delimited continuations:
  - Backtracking and search
  - Concurrency and lightweight threads
  - Probabilistic programming
  - Quantum simulation

    Scheme Pearl: Quantum Continuations. (Scheme workshop 2022)
  - Autodifferentiation and backpropagation

    Demystifying differentiable programming: shift/reset the penultimate backpropagator (ICFP '19)
  - ...

Delimited Continuations

```scala
import scala.util.continuations._
type diff = cps[Unit]

class Num(val x: Double, var d: Double) {
  def +(that: Num) = shift { (k: Num => Unit) =>
    val y = new Num(x + that.x, 0.0); k(y)
    this.d += y.d; that.d += y.d }
  def *(that: Num) = shift { (k: Num => Unit) =>
    val y = new Num(x * that.x, 0.0); k(y)
    this.d += that.x * y.d; that.d += this.x * y.d }
}

def grad(f: Num => Num @diff)(x: Double) = {
  val x1 = new Num(x, 0.0)
  reset { f(x1).d = 1.0 }
  x1.d
}

for (x <- 0 until 10) {
  assert(grad(x => x + x*x*x)(x) == 1 + 3*x*x)
}
```

Demystifying differentiable programming: shift/reset the penultimate backpropagator (ICFP '19)

First-class continuations

- We can also have undelimited first-class continuations (i.e. rest of computation up to the program end), `call/cc` in Scheme.
- Such continuations are not composable:

```
(+ 1 (call/cc (lambda (k) (begin (k 2) (k 3)))))
```

- Expressiveness: `call/cc` with mutable state can express arbitrary monadic effects and delimited continuations(Representing Monads, Filinski 1994).

Summary

- Exceptions
- Resumable exceptions (aka effect handlers)
- A family of delimited control operators
- First-class undelimited continuations
- Other considerations:
  - One-shot vs multi-shot continuations
  - Type systems ensure all effects are handled
  - Effect polymorphism
  - ...