Lecture 18: Exceptions

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Exceptions

- Note: these notes are a draft; they will be missing some things covered in lecture.
- Recall our Java example program that involves exceptoins:

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

- In this lecture we will implement an interpreter for a tiny language with exceptions. Then, we will *compile away exceptions*: we will compile our tiny language with exceptions into the untyped λ-calculus.

2 ExnLang: A Tiny Language with Exceptions

- Let's make a small language for implementing exceptions and exception handlers and implement an interpreter for this language
- Here is the abstract syntax of `ExnLang`:

```
(struct enum (n) #:transparent)
(struct eadd (e1 e2) #:transparent)
(struct eraise () #:transparent)
(struct etryhandle (etry ehandle) #:transparent)
```

- Programs in `ExnLang` evaluate to numbers. The semantics are as follows:
  - Numbers and addition evaluate as normal for this tiny language.
  - The `eraise` expression raises an exception when it is evaluated.
  - The `(etryhandle etry ehandle)` expression runs `etry`, and if an exception is raised in `etry`, it jumps to running the inner-most `ehandle` expression.
- Here are some example `ExnLang` program evaluations:
  1. `(etryhandle (eadd (enum 1) (eraise)) (enum 10))` runs to 10
  2. `(etryhandle (etryhandle (raise) (enum 30)) (enum 10))` runs to 30

3 Implementing an ExnLang Interpreter

- Now we can implement our interpreter for `ExnLang`
- Our will be for our interpreter to have two separate continuations as its argument: a *default continuation* `defaultk` that is called when no exception is raised, and a *exception continuation* that is called in the event of an exception.

```
(define (interp_h e exceptionk defaultk)
  (match e
    [(enum n) (defaultk n)]
    [(eadd e1 e2)
     (interp_h e1 exceptionk (λ (r1)
                              (interp_h e2 exceptionk
                                      (λ (r2) (defaultk (+ r1 r2))))))]
    [(eraise) (exceptionk)]
    [(etryhandle etry ehandle)
     ; run etry with a new exception handler that jumps to
     ; ehandle if an exception is encountered
     (define exceptionk2 (λ () (interp_h ehandle exceptionk defaultk)))
     (interp_h etry exceptionk2 defaultk)
     ]))
```

4 Compiling Away Exceptions

- Now we will study how to compile `ExnLang` into the untyped λ-calculus.
- Why do we do this? This gives us a way to *add exceptions to languages that do not have exceptions in them!* For example, you can add exceptions to C programs this way.
- We will use the following abstract syntax:

```
;;; type expr =
;;;   | eident of string
;;;   | elam of string * expr
;;;   | eapp of expr * expr
(struct lident (s) #:transparent)
(struct llam (id body) #:transparent)
(struct lnum (n) #:transparent)
(struct ladd (e1 e2) #:transparent)
(struct lapp (e1 e2) #:transparent)
(struct lerror () #:transparent)
```

- We have added an `lerror` expression that raises a runtime error when it is evaluated.
- To implement our compiler, we will follow a very similar structure to our ExnLang interpreter, except instead of relying on the host language (Racket) to construct and evaluate the continuation, we will emit lambda calculus terms that build and call the continuation.
- Our compiler will take 3 arguments:
  1. An `ExnLang` expression `e` to compile;
  2. A λ-calculus term `defaultk` that is the default continuation;
  3. A λ-calculus term `exceptionk` that is the exception continuation.
- We will use the notation $(\texttt{e}, \texttt{defaultk}, \texttt{exceptionk}) \rightsquigarrow l$ to denote this compilation process.
- For example, suppose we want to compile the `ExnLang` expression `(throw)`. Intuitively, this expression should compile into a λ-calculus program that calls the exception continuation. In our `ExnLang` interpreter, our exception handlers had no arguments; since our λ-calculus requires all functions to have an argument, we instead call the exception handler continuation with a default argument of 0:

$$(\texttt{(throw)}, \texttt{defaultk}, \texttt{exceptionk}) \rightsquigarrow \texttt{(lapp exceptionk 0)}$$

- `(enum n)` compiles to a λ-calculus program that calls the default continuation with an argument of `n`:

$$(\texttt{(enum n)}, \texttt{defaultk}, \texttt{exceptionk}) \rightsquigarrow \texttt{(lapp defaultk n)}$$

- The rule for exception handlers isn't too bad, but it illustrates that we can define compilation inductively using our usual horizontal-line notation:

$$\frac{\texttt{(e1, defaultk, exceptionk)} \rightsquigarrow \texttt{e1'} \qquad \texttt{(e2, defaultk, (elam "" e1'))} \rightsquigarrow \texttt{e2'}}{\texttt{(etryhandle e1 e2)} \rightsquigarrow \texttt{e2'}}$$

- Addition is the trickiest one. Here, we essentially encode our continuation-passing interpreter as a big lambda term:

- We can interpret these compilation rules as Racket code as follows:

```
(define (compile e defaultk exceptionk)
  (match e
    [(enum n)
     ; call the default handler with n
     (lapp defaultk (lnum n))]
    [(eraise)
     ; call the exception handler with 0
     (lapp exceptionk (lnum 0))]
    [(etryhandle etry ehandle)
     ; compile the handler
     (define c-handler (compile ehandle defaultk exceptionk))
     ; now compile the try block with the new handler
     (compile etry defaultk (llam "_" c-handler))]
    [(eadd e1 e2)
     ; allocate two fresh names for the arguments of the continuations
     (define r1 (fresh))
     (define r2 (fresh))
     ; build a default continuation for e1 that finishes the job of summing
     (define finish-sum (compile e2 (llam r2 (lapp defaultk (ladd (lident r1) (lident r2))))
     exceptionk))
     (compile e1 (llam r1 finish-sum) exceptionk)
     ]))
```

5 Adding Functions to ExnLang

- Let's grow our exception language by adding lambdas, so this will be our new abstract syntax:

  ```
  (struct enum (n) #:transparent)
  (struct eadd (e1 e2) #:transparent)
  (struct etryhandle (etry ehandle) #:transparent)
  (struct eraise () #:transparent)
  (struct elam (id body) #:transparent)
  (struct eapp (e1 e2) #:transparent)
  (struct eident (id) #:transparent)
  ```

- These features should all work together in the way you expect: functions behave as normally for the call-by-value λ-calculus, and an exception is handled by its inner-most handler.
- Let's look at some examples of how we expect this to behave:

6 Implementing ExnLang with Lambdas

```
;;; subst : expr -> string -> expr -> expr
;;; performs the substitution e1[x |-> e2] with lexical scope
(define (subst e1 id e2)
  (match e1
    [(eident x)
     (if (equal? x id) e2 (eident x))]
    [(elam x body)
     (if (equal? x id)
         (elam x body) ; shadowing case; do nothing
         (elam x (subst body id e2)) ; non-shadowing case
         )]
    [(eapp f arg)
     (eapp (subst f id e2) (subst arg id e2))]
    [(eadd l r)
     (eadd (subst l id e2) (subst r id e2))]
    [(etryhandle etry ehandle)
     (etryhandle (subst etry id e2) (subst ehandle id e2))]
    [(eraise) (eraise)]
    [(enum n) (enum n)]))

;;; evaluates an exnlambda term
(define (interp_h e exceptionk defaultk)
  (match e
    [(enum n) (defaultk n)]
    [(elam id x)
     (defaultk (elam id x))]
    [(eapp e1 e2)
     ; first evaluate e1, then e2
     (interp_h e1 exceptionk
               (λ (v1)
                 (interp_h e2 exceptionk
                           (λ (v2)
                             (match v1
                               [(elam id body)
                                (let* [(arg-v (interp e2))
                                       (subst-body (subst body id arg-v))]
                                  (interp_h subst-body exceptionk defaultk))])))))]
    [(eadd e1 e2)
     (interp_h e1 exceptionk (λ (r1)
                               (interp_h e2 exceptionk
                                         (λ (r2) (defaultk (+ r1 r2))))))]
    [(eraise) (exceptionk)]
    [(etryhandle etry ehandle)
     ; run etry with a new exception handler that jumps to
     ; ehandle if an exception is encountered
     (define exceptionk (λ () (interp_h ehandle exceptionk defaultk)))
     (interp_h etry exceptionk defaultk)
     ]))
```

7 Compiling Away Exceptions Again