Lecture 20: Continuation-Passing Style

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 McCarthy's Amb

- We McCarthy's `Amb` language on last week's homework. It's an interesting example of a programming language because its semantics are quite different from the usual semantics of languages we've seen.
- Here is an implementation of McCarthy's Amb:

```
(struct eif (guard thn els) #:transparent)
(struct echoose (id body) #:transparent)
(struct eand (e1 e2) #:transparent)
(struct enot (e1) #:transparent)
(struct efail () #:transparent)
(struct ebool (v) #:transparent)
(struct eident (id) #:transparent)

(define (interp-h e env failurek)
  (match e
    [(echoose id body)
     ; create a continuation that tries false and,
     ; if that fails, backtrack to try an earlier choice
     (define newk
       (λ () (interp-h body (hash-set env id #f)
                       (λ () (failurek)))))
     (interp-h body (hash-set env id #t) newk)]
    [(efail) (failurek)]
    [(ebool v) v]
    [(eident id) (hash-ref env id)]
    [(eif guard thn els)
     (if (interp-h guard env failurek)
         (interp-h thn env failurek)
         (interp-h els env failurek))]
    [(enot e) (not (interp-h e env failurek))]
    [(eand e1 e2) (and (interp-h e1 env failurek)
                       (interp-h e2 env failurek))]))

(define (interp e)
  (if (symbol? (interp-h e (hash) (λ () 'fail)))
      #f
      #t))
```

2 ExnLang with Lambdas

Recall our exception language we've been working with extended it to include lambdas, similar to your most recent homework:

```
(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)

;;; 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 e2, then e1
     (interp_h e2 exceptionk
               (λ (v2)
                 (interp_h e1 exceptionk
                           (λ (v1)
                             (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)
     ]))

(define (interp e)
  (interp_h e
            (λ () (error "uncaught exception"))
            (λ (r) r)))
```

3 Compiling Away Exceptions 1

- 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 for our λ-calculus:

```
(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.
- In Racket code, our compiler will have the following signature:

```
(define (compile e defaultk exceptionk)
  (match e
    ...))
```

- The correctness criteria for our compiler is the usual one: Suppose `ExnLang` program e runs to a value v_1 and e compiles to a λ-calculus program e'. Then, it should be the case that e' runs to a value that is "the same as" v_1, for an intuitive definition of "same as" (i.e., numbers are compared for equality, functions are compared for extensional equality).
- Let's go through the compilation cases one at a time, and check that each one validates this correctness condition.

4 Compiling Away Exceptions 2: Basic Cases

- The key idea of this compiler is to essentially translate our continuation-passing interpreter for `ExnLang` into the λ-calculus.
- 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.
- Similarly, to compile away numbers, we call the default continuation with the number as an argument.
- Translating these two cases into code:

```
(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))]
    ...))
```

- Continuing on, let's compile `etryhandle`. Again, this compilation looks quite similar to how we implemented our continuation-passing interpreter:

```
(define (compile e defaultk exceptionk)
  (match e
    ...
    [(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))]
    ...))
```

- With these 3 features we can begin testing our compiler to make sure that it works as we expect on some small examples. First, let's define an auxiliary function that compiles an `ExnLang` program and runs it using the λ-calculus interpreter:

```
(define (compile-and-run e)
  (interp-l (compile e
      (llam "x" (lident "x")) ; initial default continuation is identity function as usual
      (llam "x" (lerror)))))  ; initial exception handler raises a Racket runtime error
```

- It's worth pausing to examine the output of this compiler on a simple case:

```
> (compile (etryhandle (eraise) (enum 20)) (llam "x" (lident "x")) (llam "x" (lerror)))
(lapp
   (llam "_"
     (lapp (llam "x" (lident "x")) (lnum 20))) (lnum 0))
```

- Ask yourself: which parts of this return result come from which parts of the compiler?
- Now we can run some basic tests:

```
> (check-equal? (compile-and-run (enum 10)) (lnum 10))
> (check-equal? (compile-and-run
    (etryhandle (eraise) (enum 20))) (lnum 20))
```

5 Compiling Away Exceptions 3: Addition

- Now let's start to handle some trickier operations. First, addition. The rule for compiling addition looks very similar to how we implemented addition in our continuation-passing interpreter, except we are using λ-calculus terms to do the job of calling continuations instead of the Racket host language:

```
(define (compile e defaultk exceptionk)
  (match e
    [(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)]
    ...))
```

- Let's see some small examples of how this works:

```
> (compile (eadd (enum 10) (enum 20)) (llam "x" (lident "x")) (llam "x" (lerror)))
  (lapp
   (llam "23" (lapp (llam "24" (lapp (llam "x" (lident "x")) (ladd (lident "23") (lident
    "24")))) (lnum 20)))
   (lnum 10))
```

- This is a fairly big result! Let's step through it by hand to see how it adds these two numbers:

```
(lapp
  (llam "23" (lapp (llam "24" (lapp (llam "x" (lident "x")) (ladd (lident "23") (lident
    "24")))) (lnum 20)))
    (lnum 10))
-- substitute 10 in for identifier "23" -->
(lapp (llam "24" (lapp (llam "x" (lident "x")) (ladd 10 (lident "24")))) (lnum 20))
-- substitute 20 in for identifier "24" -->
(lapp (llam "x" (lident "x")) (ladd 10 20))
-- evaluate identity -->
30
```

- So, we got the result we expected, and we can see the point when the default continuation (the identity function) was called.

- We can test that addition works as expected:

```
> (check-equal? (compile-and-run (eadd (enum 10) (enum 20))) (lnum 30))
> (check-equal? (compile-and-run
      (etryhandle (eadd (enum 10) (eraise))
            (enum 20))) (lnum 20))
```

6 Compiling Away Exceptions 4: Lambdas and Function Calls

- Lambdas and function are the trickiest case because *you do not know exception handler will be present when a function is called*. For example, the exact same function can be called twice within two different exception handlers, and it should have a different behavior in both of those cases. Here is a small example of such a program:

```
> (define t7
  (eapp (elam "f"
              (eadd (etryhandle (eapp (eident "f") (enum 0)) (enum 30))
                    (etryhandle (eapp (eident "f") (enum 0)) (enum 50))))
        (elam "x" (eraise))))
> (check-equal? (interp t7) 80)
```

- Because of this, when we are compiling lambdas, we will need to add extra arguments for the default continuation and exception continuation, and compile the body of the lambda to account for these. The intuition is that the default and exception continuation will be provided at the call-site of the lambda.

```
(define (compile e defaultk exceptionk)
  (match e
    [(elam id body)
    ; generate a lambda term that takes a default continuation and a handler continuation as
     an arg
    (define defaultk-name (fresh))
    (define exceptionk-name (fresh))
    (define compiled-lambda (compile body (lident defaultk-name) (lident exceptionk-name)))
    (lapp defaultk (llam defaultk-name
                         (llam exceptionk-name
                               (llam id compiled-lambda))))]
    ...))
```

- Then, to compile application, we need to pass the current default and exception continuation in as an argument to the lambda:

```
(define (compile e defaultk exceptionk)
  (match e
  [(eapp e1 e2)
     (define r1 (fresh))
     (define r2 (fresh))
     (define callfunc
       (compile e1
                (llam r1
                      (lapp (lapp (lapp (lident r1) defaultk) exceptionk) (lident r2)))
    exceptionk))
     (compile e2 (llam r2 callfunc) exceptionk)]
   ...))
```

- This compilation is pretty similar to addition: first, it evaluates `e1` and `e2` and binds the results of these evaluations to `r1` and `r2` respectively. Then, it calls `r1` 3 times in a row: first, with the current default continuation; then, with the current exception continuation; then, finally, with the result of evaluating `e2`.

- Now we can test this:

```
> (check-equal? (compile-and-run t7) (lnum 80))
```