CIS 425: Principles of Programming Languages
Lecture 14: Exception & Continuation & CPS

Xiaodong Quan

22 May 2019

Today's lecture was focused on Exception, Continuation and Continuation Passing Style (CPS). Firstly, this note will refer some important info from textbook, then will head in lecture contents:

- Important TextBook Info (see Section 1)
- Exception from Lecture (see Section 2)
- Continuation from Lecture (see Section 3)
- Continuation Passing Style from Lecture (see Section 4)

1 Textbook

You will see some refer of Structured Control Background Info, and general introduction of Exception in this section.

- **Section 1: Structured Control & Background Info**

  "If you scan the code from top to bottom, you might get the idea that the instructions between labels 10 and 20 act together to perform some meaningful task. However, then as you scan downward, you can see that it is possible later to jump to instruction11, which is in the middle of this set of instructions."

  (GOTO statement just like we learned in 314, Jump to code with target Label.)

```
Label: 10 IF (X .GT. 0.000001) GO TO 20
X = -X
Label: 11 Y = X*X – SIN(Y) / (x+1)
IF (X .LT. 0.000001) GO TO 50
Label: 20 IF (X*Y .LT. 0.000001) GO TO 30
X = X + Y
Label: 30 X = X+Y
...
Label: 50 CONTINUE
X = A
Y = B-A + C*C
GO TO 11
```

"In modern programming style, we group code in logical blocks, avoid explicit jumps except for function returns, and cannot jump into the middle of a block or function body."

```
If ... then ... else ... end
while ... do ... end
for ... { ... }
case ...
```

"The restriction on jumps into blocks illustrates the value of leaving a construct out of a programming language. If a label is placed in the middle of a function body and a program executes a jump to this label, what should happen? Should an activation record be created for the function call? If not, then local variables will not be meaningful. If so, then how will function parameters stored in the activation record be set? Without executing a call to the function, there are no parameter values to use in the call. Because these questions have no good, convincing, clear answers, it is better to design the compiler to reject programs that might jump into the middle of a function body."

• Section 2: Exceptions (General Intro)

Purpose of an Exception Mechanism:

1. Jump out of a block or function invocation.
2. Pass data as part of jump.
3. Return to a program point that was set up to continue the computation.

Every exception mechanism includes two constructs:

1. A statement or expression form for raising an exception, which aborts part of the current computation and causes a jump (transfer of control).
2. A handler mechanism, which allows certain statements, expressions, or function calls to be equipped with code to respond to exceptions raised during their execution.

Note:
1. Raising an exception also be called as Throwing an exception.
2. Handling an exception also be called as Catching an exception.

Why exception have become an accepted language construct?

1. Many languages do not otherwise have a clean mechanism for jumping out of a function call, for example, aborting the call.
2. Rather than using go to statement, which can be used in unstructured ways, many programmers prefer exceptions, which can be used to jump only out of some part of a program, not into some part of the program that has not been entered yet.
3. Exceptions also allow a programmer to pass data as part of the jump, which is useful if the program tries to recover from some kind of error condition.
4. Exceptions provide a useful dynamic way of determining to where a jump goes. If more than one handler is declared, the correct handler is determined according to dynamic scoping rules.

- ML Exceptions

Three Parts of ML mechanism:

1. Exceptions declarations, which declare the name of an exception and specify the type of data passed when this exception is raised.
2. A raise expression, which raises an exception and passes data to the handler.
3. Handler declarations, which establish handlers.

ML Exception form:

```
exception <name> of <type>
```

<name> - The Name of the Exception
<type> - The type of data that are passed to an exception handler

Note: of <type> is optional part, take two examples as below:

```
exception Ovflw;
exception Signal of int;
```

(Exception Ovflw doesn't needs any input data, but Signal need an input with type int.)

ML Raise form:

```
<name> - A previously declared exception name
<arguments> - Should have a type that matches the exception declaration
```

Example:

```
raise Ovflw;
raise Signal (x+4);
```

(Type was declared before is Int, so here the argument -¿ (x+4) matches type Int.)

ML Handler form:
An ML handler is part of an expression form that allows a handler to be specified. An expression with handler has the form.

```
<exp1> handle <pattern> => <exp2>;
```

Above expression will evaluated by following order:
Evaluate the entire expression by evaluate <exp1>;

```
IF Evaluation of <exp1> terminates normally (no error)
        - Return the result returned by <exp1>
ELSE IF <exp1> raises an exception that matches <pattern>
        - Any values passed in raising the exception are bound according to
        <pattern>
        - Evaluate <exp2>
            IF <exp2> raises an exception
                - GO TO CASE (GO TO will jump to target immediately!)
        - Return the result returned by <exp2>
ELSE IF <exp2> raises an exception that does not matches <pattern>
        - GO TO CASE
Label: CASE
        - Means the entire expression has an uncaught exception that can be
        caught by the handler established by an enclosing expression or function
        all.
        - More details later.
```

Example 1:

```
exception Ovflw;                    /* Declare the Exception Name */

fun f(x) = if x <= Min              /* Declare the Exception Function */
        then raise Ovflw
        else 1/x
end

/* Use two different way to handle Ovflw in Expression */

( f(x) handle Ovflw => 0 ) / ( f(x) handle Ovflw => 1 )
```

Note:
In the numerator, the handler returns value 0. In the denominator it will cause division by zero if the handler still return value 0; In this case, the handler will returns value 1 if the Ovflw exception is raised.

Example 2:

```
exception Signal of int;

fun f(x) =
        if        x = 0 then raise Signal (0)
        else if x = 1 then raise Signal (1)
        else if x = 10 then raise Signal (x-8)
        else (x-2) mod 4;

 f (10) handle Signal (0) => 0
                 | Signal (1) => 1
                 | Signal (x) => x + 8;
```

The handler in this expression uses pattern matching, which follows the form established for ML function declarations. More specifically, the meaning of an expression of the form

```
<exp> handle <pattern1> => <exp1>
             | <pattern2> => <exp2>
```

2 Exception

- **Why use Exception?**
  we have below function:

```
- fun prod nil = 1;
|       prod (x :: xs) = x * (prod xs);
```

  Suppose we have a 100000...0000 long list, which looks like [1,2,0,4, ... ,5]
  What we got would be (1 * 2 * 0 * ... * 5)
  We know what ever it is after the 0, our result will still be 0;
  In this case, we are wasting memory to make evaluation;
  So, we want an exception to stop useless calculation in this situation.

  Now, let's optimize above function:

```
- exception ZERO          \* declare an Exception *\

- fun prod' nil = 1;
|       prod' (x :: xs) = x * (prod xs);
|       prod' (0 :: xs) = raise ZERO;
```

  Now, if we call the function as below with exception handle:

```
- prod [1, 2, 0, ... , 5];
- fun prod xs = (prod' xs) handle ZERO => 0;
```

  Above function will return 0 once it catch exception named ZERO;
  Now we won't waste extra memory since evaluation will terminate once ZERO been handled;

- **Simple example for use of exception**
  Base on function declaration below:

```
- exception E of int;

- fun f x = raise E x;
```

  1. What is the result of below Code?

```
((f 0) handle E x => 99) + ((f 5) handle E x => 100)
```

  \-   result is 99 + 100 = 199

  2. What is the result of below Code?

```
(f 0) + (f 5);
handle E x => x;
```

  \-   result is 0 + 5 = 5

- **Scope of Exception:**
  Take a look at below Pseudo code:

```
exception x

( let
.    fun f y = raise x
.    fun g h = (h 1) in handle x => 2
.    (g f) handle x => 4
. end
) handle x => 6
```

What is the result?
Note: Exception always have Dynamic Scope

$\boxed{x=6}$

$\boxed{f}$

$\boxed{g}$

$\boxed{x=4}$

$\boxed{\begin{array}{l}h\\x=2\end{array}}$ g(f)

$\boxed{y=1}$ h 1

In this case, when fun f y raised exception x, it will go to find the nearest handle. So the handle x => 2 will be find. In this case, the result is **2**;

But how about we remove the handle x => 2 part?

```
exception x

( let
.    fun f y = raise x
.    fun g h = (h 1)
.    (g f) handle x => 4
. end
) handle x => 6
```

What is the result?

$\boxed{x=6}$

$\boxed{f}$

$\boxed{g}$

$\boxed{x=4}$

$\boxed{h}$ g(f)

$\boxed{y=1}$ h 1

Then the result would be **4** since the nearest handle have x => 4;

- **Exception in Typing problem:**
  Suppose we have function below:

```
- fun head (x :: xs) = x;
```

  How should we handle with the input as nil?
  Let's see some Examples based on the current function we have:

```
head [1, 2, 3] => 1          (int -> int)
head [true, false] => true          (bool -> bool)
head nil => error
```

  Assume: could we just set a sign means we got a nil head? like:

```
- fun head (x :: xs) = x;
|       head nil = 99;
```

  Can we just do that? if we got 99 then means we got nil head? **No!**

  Why?

  Because if we do that, which means we set a return type on function head. Now, the return type only could be int. In other words, when we trying to run head [true, false], it will cause error, since the input type must be int now as well!

```
head [1, 2, 3] => 1          (int -> int)
head [true, false] => Type Error
head nil => 99
```

  Note: raise Exception alway have type α

  So, let's optimize former function we have:

```
- exception HeadEmpty
- fun head (x :: xs) = x;
|       head nil = raise HeadEmpty;
```

  Let's recall former examples we have:

```
head [1, 2, 3] => 1          (int -> int)
head [true, false] => true          (bool -> bool)
(head nil handle HeadEmpty => 99) => 99
```

  If we use handle to catch the exception, and return what we need, it works!

3 Continuation

- **What is Continuation?**
  Continuations are a programming technique, based on higher-order functions, that may be used directly by a programmer or may be used in program transformations in an optimizing compiler.

- **Example based on Left to Right evaluator:**

  Example:

  (2 + 3) + (4 + 1)

  Step1: (2 + 3) => v
  Step2: v + (4 + 1) = (fn v => v + (4 + 1))

  Now, we say
  $\boxed{\text{v + (4 + 1)}}$ and $\boxed{\text{fn v => v + (4 + 1)}}$ both Continuation of $\boxed{\text{(2 + 3)}}$

- **Example based on Right to Left evaluator:**

  Example:

  (2 + 3) + (4 + 1)

  Step1: (4 + 1) => v
  Step2: v + (2 + 3) = (fn v => v + (2 + 3))

  Now, we say
  $\boxed{\text{v + (2 + 3)}}$ and $\boxed{\text{fn v => v + (2 + 3)}}$ both Continuation of $\boxed{\text{(4 + 1)}}$

4 Continuation Passing Style

- **What is CPS?**

  There is a program form called Continuation-Passing Style in which each function or operation is passed a continuation. This allows each function or operation to terminate by calling a continuation. As a consequence, no function needs to return to the point from which it was called.

  This property of continuation-passing form may remind you of tail recursive, as a tail recursive need not return to the calling function.

- **Not Tail Recursive & Not CPS Example:**

```
- fun length nil = 0
|      length (x :: xs) = 1 + length xs
```

  Based on the Definition of CPS and Tail Recursive,
  Above function Not CPS and Not Tail Recursive.
  Since it need return to it caller after finished recursive.

- **CPS Example:**

```
- fun length' nil k = k 0
|      length' (x :: xs) k = length' xs (fn v => k (1 + v))
```

```
length' (1 :: 2 :: nil) (fn x => x)

length' (2 :: nil) (fn v => (fn x => x) (1 + v))
length' (2 :: nil) (fn v => (1 + v))

length' (nil) (fn v => (fn v => (1 + v)) (1 + v))
length' (nil) (fn v => (1 + (1 + v)))

(fn v => (1 + (1 + v))) 0

(1 + (1 + 0))

2
```

  Above is the CPS Example uses to calculate the length of a list.