

CS421 Lecture 20: Control Flow¹

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

July 25, 2006

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Overview

Continuations

Exceptions

Coroutines

Objectives

By the end of this lecture, you should

- ▶ be familiar with different kinds of control flow constructs
- ▶ understand the basics of continuations and the Scheme `call/cc` construct
- ▶ understand exceptions, especially the approach taken by modern OO languages like Java
- ▶ know about coroutines at a high-level, including how they work and what they can accomplish

Classes of Control Flow

We can break control flow up into a number of categories, based on the purpose of the constructs.

- ▶ Invocation
 - ▶ Direct calls: functions, subroutines
 - ▶ Indirect calls: function pointers, class methods, closures
- ▶ Termination of Scope
 - ▶ Structured: break, break to a label, exceptions, CPS
 - ▶ Unstructured: goto, setjmp/longjmp, exit
- ▶ Selection
 - ▶ Structured: if/then/else, continue, switch, case
 - ▶ Unstructured: goto, computed goto, labeled entries

Classes of Control Flow, cont.

- ▶ Iteration
 - ▶ Precomputed iteration space: do, foreach
 - ▶ Dynamic iteration space: for, while, recursion
- ▶ Concurrency
 - ▶ Manual: processes, threads, futures, coroutines
 - ▶ Automatic: constructs in concurrent/parallel frameworks for reductions
 - ▶ Communication and synchronization techniques are critical
- ▶ You should have come across many of these already:
if/then/else, switch/case, do, for, while, break, continue,
probably foreach, goto, exit

Continuations

At its core, the continuation is just “the rest of the computation”
– it tells us what we have left to do.

- ▶ Continuations can be used to model many control flow constructs
- ▶ Continuation manipulation is a fundamental feature of Scheme
- ▶ Also, very important to denotational semantics; continuations allow modelling of complex control flow

Call/CC

To do something interesting with a continuation, we first need to get ahold of one. In Scheme, this is done using the `call/cc` function.

- ▶ The continuation is represented as a function expecting one argument
- ▶ The `call/cc` call also expects a function that takes one argument as its argument
- ▶ When `call/cc` is called, the current control context is saved and passed to the function argument of `call/cc`
- ▶ Invoking this argument then returns control to where we were when we invoked `call/cc`

Call/CC Example 1

```
1 (define call/cc call-with-current-continuation)
```

The full name is a bit cumbersome, but some Schemes don't define the abbreviation...

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

- ▶ The continuation is `(+ 1 [])`, where `[]` represents the “hole” where the continuation result will be placed
- ▶ The final result is just 4 `(1 + 3)`, since we never add 2 to the result.

Call/CC Example 2

```

1 (define list-product
2   (lambda (s)
3     (call/cc
4       (lambda (exit)
5         (let recur ((s s))
6           (if (null? s) 1
7               (if (= (car s) 0) (exit 0)
8                   (* (car s) (recur (cdr s))))))))))
9 > (list-product '(1 2 3 4 5))
10 120
11 > (list-product '(1 2 0 3 4 5))
12 0

```

This allows us to “jump out” when we find a 0, meaning that the product of the entire list is 0

Continuations, Overall

- ▶ The good: Using continuations with `call/cc` gives us a flexible way to encode jumps, meaning we could build up features like exceptions.
- ▶ The bad: This could get cumbersome, though – if we have a dedicated exception mechanism, even just syntactic sugar, it would make programs clearer.

Error Handling

Historically, languages and language APIs have handled errors in several different ways:

- ▶ Have functions return values, outside of the normal range of return values, that represent errors;
- ▶ Return a success or error code from the function, requiring actual return values to be in pointers or reference parameters;
- ▶ Pass an error handling routine (like a continuation) that the normal routine can invoke when errors occur.

Special Return Values

```
1 if ((he=gethostbyname(hostname)) == NULL) {
2     perror("clientConnect: failed to resolve ip address");
3     return HOSTNOTFOUND;
4 }
5
6 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
7     perror("clientConnect: failed to create socked");
8     return OTHERERROR;
9 }
```

Success or Error Returns

```
1 pthread_mutex_t theLock = PTHREAD_MUTEX_INITIALIZER;
2 ...
3 if (pthread_mutex_lock(&theLock) != 0) {
4     perror("Could not acquire lock");
5     return OTHERERROR;
6 }
7 ...
8 if (pthread_mutex_unlock(&theLock) != 0) {
9     perror("Could not release lock");
10    return OTHERERROR;
11 }
```

Are these good enough?

These methods work, but all tend to be clumsy.

- ▶ Using return codes require picking values that can never be returned from the function, and are easy to forget to check
- ▶ Having explicit return codes for success or failure again means these have to be checked, which clutters up the code and leads to “unnatural” use of functions
- ▶ Error handlers again clutter up function calls, and can be expensive

Are there any better solutions?

Are these good enough?

These methods work, but all tend to be clumsy.

- ▶ Using return codes require picking values that can never be returned from the function, and are easy to forget to check
- ▶ Having explicit return codes for success or failure again means these have to be checked, which clutters up the code and leads to “unnatural” use of functions
- ▶ Error handlers again clutter up function calls, and can be expensive

Are there any better solutions?

Exceptions

Exceptions

- ▶ Provide for “out-of-band” error handling, with error handlers moved into special sections of code
- ▶ Eliminates the need to check for errors with each step of computation
- ▶ Often allows for exception mechanisms that fit well with the strengths of the language (exception objects in Java, constructor-based exceptions in OCaml, etc)

Exception History

Exceptions were first introduced in PL/1, or “Programming Language 1”, an IBM language from the 1960’s.

- ▶ Used an exception handler of the form *ON condition statement*, which would execute the handler statement if the error condition matched.
- ▶ For very serious errors, would terminate the program after handling the error
- ▶ For less serious errors, would pick up execution at the statement following the one that triggered the exception (similar to mechanisms in Visual Basic)
- ▶ Wound up being confusing – hard to tell, from looking at code, what would execute after an error occurred

Modern Exception Mechanisms

Many languages now include exceptions.

- ▶ Ada includes a built-in type `exception` to represent exceptions;
- ▶ Modula-3 provides a separate top-level “kind” of entity, like a type or constant, to represent exceptions
- ▶ Many OO languages (Java, C++, C#, Python) just use an object to represent an exception
- ▶ ML provides exception constructors, similar to type definitions
- ▶ Most languages allow parameters in the exceptions, allowing for passed values, messages, etc.

Exceptions in Java

In Java, exceptions are objects like other objects in the system, and are joined into the same inheritance hierarchy.

- ▶ Exceptions all extend class `Throwable`
- ▶ Most exceptions extend `Exception`, which is for *checked* exceptions
- ▶ System exceptions extend `RuntimeException` or `Error`, which are unchecked (do not need to be caught)
- ▶ Exceptions can be manually thrown with the `throw` statement

Example: Java Exception Handler

```
1 ParallelTerm currentTerm = null;
2 try {
3     currentTerm = (ParallelTerm)definition.clone();
4 } catch (Exception e) {
5     System.err.println("Clone exception caught!");
6     e.printStackTrace();
7     System.exit(-1);
8 }
```

Multiple Exception Handlers

Exceptions in Java can have multiple exception handlers.

- ▶ Handlers are tried in order, with the first handler defined for the runtime class of the exception or a parent of this class being used
- ▶ Handlers for class `Exception` act as “else” cases, catching any checked exception (and thus should come last in the list!)
- ▶ Uncaught exceptions must be listed in a `throws` clause of the method

Example: Multiple Handlers and throw

```
1 try {  
2     ...  
3 } catch (IOException e) {  
4     ...  
5     throw new MigrateActorException(e.getMessage());  
6 } catch (ClassNotFoundException e) {  
7     ...  
8     throw new MigrateActorException(e.getMessage());  
9 };
```

Exception Propagation

Exceptions work by “unwinding” the call stack until a proper handler is found.

- ▶ In languages like C++, destructors for local objects must be called
- ▶ Work typically done on function returns must be performed (saved registers are set back to the proper values, stack space is deallocated, etc)
- ▶ This unwinding may unwind all the way back to the start of the program – this generally causes a default exception handler surrounding the start of the program to be invoked

Forcing Cleanup

Some languages include a `finally` block as part of exceptions:

- ▶ Used to indicate cleanup code that should always run at the end of the exception handler block;
- ▶ called either when exception handler ends or before the exception propagates up the stack;
- ▶ very complex semantics, since exceptions and finally blocks can also throw exceptions.

Finally Example

This example is from p.448 of Scott's Programming Language Pragmatics:

```
1  FileReader myStream = null;
2  try {
3      myStream = new FileReader(new File("foo"));
4      parse(myStream);
5  } catch (EOFException e) {
6      System.out.println("Oops; input file too short.");
7  } finally {
8      myStream.close();
9  }
```

Question: Are Checked Exceptions a Good Idea?

One debate over exceptions is whether having checked exceptions is a good idea.

- ▶ Java says yes – checked exceptions are valuable, since a programmer has to deal with any possible exceptions
- ▶ C++ and C# say no – checked exceptions lead to worthless exception handlers that obfuscate the real source of exceptions, since people just put in wrappers that “eat” the exception to make the language happy

Coroutines

- ▶ Similar to continuation – represented as a closure with a code location and referencing environment
- ▶ Call of coroutine is a non-local transfer of control (similar to a goto, but more structured); referred to as `transfer`
- ▶ Difference with continuation – code location can change with each call, so when we transfer we keep track of where we were last

Coroutines

Coroutines basically give us multiple execution contexts existing concurrently but only running one at a time.

- ▶ Can be used to model other control abstractions (iterators, threads)
- ▶ Can be used for situations where we want to maintain program state between calls to the coroutine
- ▶ Well suited to discrete event simulation

Coroutine Example: Beta

```
1 (# i: @Integer;  
2   b: @| (# do (L: i+5->i; suspend;  
3             restart L  
4             :L)  
5             exit i  
6             #)  
7   do 10->i; b->i; i->putint;  
8       b->i; i->putint  
9 #)
```

Here, we can call `b` as an incrementer, which will add 5 to `i` each time it is called, and will remember where it left off in the code (hence the need to `restart`).

Couroutine Example 2: Beta

```

1  (# Factorial: @|
2    (# T: [100] @ Integer; N,Top: @Integer;
3    enter N
4    do 1->Top->T[1];
5    Cycle(#
6    do (if (Top<N) // True then
7        (Top+1,N)->ForTo
8        (#do T[inx-1]*i->T[inx]#);
9        N->Top
10       if);
11       N+1->N; SUSPEND;
12     #)
13     exit T[N-1]
14   #);
15   F: @Integer
16   do 4->Factorial->F; Factorial->F; 3->Factorial->F;
17 #)
  
```

Coroutine Internals

- ▶ Coroutine execution, like normal program execution, occurs in the context of an execution stack and system state (register settings, for instance)
- ▶ Transfer can be seen as saving all this information and replacing it with either new information (a new coroutine starting) or previously saved information
- ▶ When coroutine is reentered, state is set back to what it was on transfer out and process can pick up where it left off

This is similar to a PL version of multi-tasking.