CnC-Scala

A Scala based Implementation of CnC

January 18, 2012

Shams Imam

Rice University

CnC-Scala

- Written in pure Scala
  - uses continuations compiler plugin
- Only dependency – jsr-166y.jar
  - comes bundled with java 7
- Supports both Continuation and DDF policies
- Distribution with examples available at:
  - http://cnc-scala.rice.edu/

Background

- Current CnC Execution Policies
  - Different Blocking schemes
    - Allocates extra threads
      - Context switching overhead
      - memory issue
  - Rollback and Replay
    - Possible to have too many replays
    - Need to ensure replays are side-effect free

Background (contd.)

- Current CnC Execution Policies...
  - Delayed Async
    - Re-evaluation of guards
    - Code duplication (isReady() and compute())
  - Data-Driven Futures (DDFs)
    - Only Tag-dependent gets
    - Code duplication (createAwaitsList() and compute())

Alternate policy

- What to do when get() has unavailable items?
  - Use thread-independent delimited one-shot continuations
- Benefits:
  - Allow arbitrary (data-dependent) gets
  - No code duplication

Continuations

- Represents rest of the computation from a given point in the program
- Allows
  - **suspend**ing current execution state
  - **resume** from that point later

Continuations (contd.)

- Delimited Continuations
  - Limits the region of code represented in the continuation (of interest to us)
  - Can return a value (of interest to us)
  - Can be composed (not of interest to us)
  - e.g. shift-reset in Scheme
- One-shot Continuations
  - Continuations that are invoked only **once**
  - Normally continuations are multi-shot

Benefits of using continuations

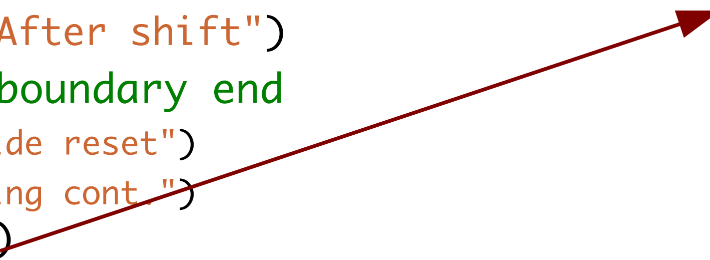
- No re-execution of code
  - Unlike rollback and replay
- Allow arbitrary (data-dependent) gets
  - Unlike DDF
- No code duplication
  - Unlike delayed async, DDF
- No extra threads
  - Unlike blocking policies

JVM Continuation frameworks

- Javaflow
  - Simulates bytecode operand stack
- Scala continuations
  - CPS transforms code
  - Inspired from Scheme's shift/reset
- Kilim continuations
  - Reportedly one of the faster implementations
  - Only transforms suspendable methods

Scala Continuation example

```scala
object Main {
  def main(args: Array[String]) {
    println("1. Main starts")
    var continuation: (Unit) => Any = null
    reset { // delimit boundary start
      println("2. Entered reset")
      shift[Unit, Any, Unit] {
        delimCont =>
          println("3. Entered shift")
          continuation = delimCont
          println("4. Inside shift")
      }
      println("5. After shift")
    } // delimit boundary end
    println("6. Outside reset")
    println("7. Calling cont.")
    continuation()
    println("8. Main ends")
  }
}
```



Output:

```
1. Main starts
2. Entered reset
3. Entered shift
4. Inside shift
6. Outside reset
7. Calling cont.
5. After shift
8. Main ends
```

CnC-Scala Runtime - step

- Wrap each step.compute() in a reset

```
// some preparation
reset {
  step.compute(tag, inputs, outputs)
}
// execution logic based on
// whether continuation was stored
```

CnC-Scala Runtime - get

- If item available return it
- Else store continuation

```
get(tag: TagType): ItemType = {
  if (itemAvailable)
    return item
  else
    shift { continuation =>
      // store continuation
    }
    // when cont resumes, item is available
    return item
}
```

CnC-Scala Runtime - put

- Store item
- Resume waiting continuations

```
put(tag: TagType, item: ItemType) {
  // store item
  // resume ALL waiting continuations
}
```

CnC-Scala User Flowchart

CnC Graph in Scala

cnc_scala_translator

Stub code in Scala

User writes:
- Step code in Scala
- Main method in Scala

cnc_scala_compile

Compiled bytecode

cnc_scala_run

Output

CnC-Scala Demo

- Show demo or youtube link
  - http://www.youtube.com/watch?v=1Ya4S8diezs
- Steps:
  1. Write CnC graph description
  2. Use translator to generate Stub Code
  3. Write Step implementations and main()
  4. Compile and run code

CnC-Scala Results

- Machine Specification:
  - 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4 GHz
  - 32 GB memory
  - 32 kB L1 cache and a 3 MB L2 cache (per core)
  - Sun Hotspot JDK 1.6 (default settings)
  - Scala 2.9.1.final
  - Minimum execution time of 5 runs reported
    - Each run includes GC and JVM warm-up time
  - Cholesky 2000 x 2000 with tile size of 250 x 250

Results – Cholesky Exec Time

Comparison of different implementations

Execution Time (ms)

8000
7000
6000
5000
4000
3000
2000

2 4 6 8 10 12 14 16

Number of Worker Threads

Scala-ddf Scala-cont Javaflow

Results – Cholesky Activity Count

Conclusions and Future Work

- Performance is acceptable, given we
  - relax restrictions on get
  - avoid code duplication
- Create hybrid policy:
  - DDF + Continuation mode
- Improve runtime to reduce cost of running activities
  - Avoid debug code overheads, e.g. atomic counters
  - Reduce ThreadLocal use
- Try out Kilim-inspired implementation

Thank you!

[image source: http://www.jerryzeinfeld.com/tag/question-of-the-day/]